High Performance JavaScript
Build Faster Web Application Interfaces

高性能 JavaScript

[美] *Nicholas C. Zakas* 著

丁琛 译 赵泽欣 审校

O'REILLY®

YAHOO! PRESS

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

JavaScript/Web开发

高性能JavaScript

如果你像大多数开发者那样，高度依赖JavaScript开发可交互且快速响应的网络应用，那么JavaScript代码会让你的应用变得缓慢。本书揭示的技术和策略能够帮助你在开发中突破性能瓶颈。你将会学到如何缩短执行时间、提高加载速度、改善DOM交互、优化页面生存周期，等等。

雅虎的前端工程师Nicholas C. Zakas和其他五位JavaScript专家——Ross Harmes、Julien Lecomte、Steven Levithan、Stoyan Stefanov、Matt Sweeney，演示了页面加载代码的最佳方案，并且介绍了让JavaScript尽可能高效执行的编程技巧。你将会学到将文件打包部署到生产环境的最佳实践，以及能够帮助你排查线上问题的工具。

- 找出有问题的代码并给出更优替代方案
- 理解JavaScript存取数据的原理，改善代码
- 改善JavaScript代码来加速DOM交互
- 使用优化技术来改善执行性能
- 学习多种方式以确保UI一直处于可用状态
- 实现更快的客户端与服务端通信
- 使用打包系统精简文件，并使用HTTP压缩传输

“《高性能JavaScript》涵盖了当今JavaScript开发者需要了解的所有性能问题，毫无疑问，它已加入我的性能最佳实践列表。”

——Steve Souders

“《高性能JavaScript》是个让人印象深刻的JavaScript话题、技巧、秘诀的集合。如果你想编写高质量 JavaScript 代码，这本书值得一读。”

——Venkat Udayasankar
雅虎搜索性能专家

Nicholas C. Zakas，雅虎首页的主要开发者，雅虎用户界面库（YUI）代码贡献者，擅长利用JavaScript、HTML、CSS、XML、XSLT设计和实现WEB界面的软件工程师。

推荐中高级JavaScript开发者阅读。

图书分类：Web开发

策划编辑：张春雨
责任编辑：徐津平

O'REILLY®
oreilly.com

Broadview®
WWW.BROADVIEW.COM.CN

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-121-26677-5

9 787121 266775 >

定价：65.00元

高性能

JavaScript

[美] *Nicholas C. Zakas* 著

丁琛 译 **赵泽欣** 审校

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

如果你使用 JavaScript 构建交互丰富的 Web 应用，那么 JavaScript 代码可能是造成你的 Web 应用速度变慢的主要原因。本书揭示的技术和策略能帮助你在开发过程中消除性能瓶颈。你将会了解如何提升各方面的性能，包括代码的加载、运行、DOM 交互、页面生存周期等。雅虎的前端工程师 Nicholas C. Zakas 和其他五位 JavaScript 专家介绍了页面代码加载的最佳方法和编程技巧，来帮助你编写更为高效和快速的代码。你还会了解到构建和部署文件到生产环境的最佳实践，以及有助于定位线上问题的工具。

© 2010 by O'Reilly Media, Inc.
Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2015. Authorized translation of the English edition, 2010 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.
All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有出版权由 O'Reilly Media, Inc. 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号　图字：01-2010-4394

图书在版编目（CIP）数据

高性能 JavaScript /（美）泽卡斯（Zakas,N.C.）著；丁琛译. —北京：电子工业出版社，2015.8
书名原文：High Performance JavaScript
ISBN 978-7-121-26677-5

Ⅰ. ①高… Ⅱ. ①泽… ②丁… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2015)第 162448 号

策划编辑：张春雨
责任编辑：徐津平
封面设计：Karen Montgomery　张　健
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：14.5　字数：344 千字
版　　次：2015 年 8 月第 1 版
印　　次：2015 年 8 月第 1 次印刷
定　　价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

谨以此书献给我的家人，妈妈、爸爸和Grey，
是你们的爱和支持使我走过这些岁月。

译者序

这是一本关于 JavaScript 性能的书。

在 Web 应用日趋丰富的今天，越来越多的 JavaScript 被运用在我们的网页中。随着用户体验被日益重视，前端性能对用户体验的影响开始备受关注，而引起性能问题的因素相对复杂，因此它很难得到全面的解决。这本书是一个契机，它尝试着从多个方面综合分析导致性能问题的原因，并给出适合的解决方案，帮助我们改善 Web 应用的品质。

这本书页数不多，但它承载着 JavaScript 性能方面最为宝贵的经验。不仅从语言特性、数据结构、浏览器机理、网络传输等层面分析导致性能问题的原因，还介绍了多种工具来帮助我们提升开发过程和部署环节的工作效率。

本书作者 Nicholas C. Zakas 是一位经验丰富的前端专家，他的许多研究（*www.nczonline.net*）对前端业界的贡献让我们受益匪浅。本书的另外五位特约作者均为各自领域的专家，他们的专业技能和知识的融入使得本书内容更为充实，更具实用价值。

特别感谢赵泽欣（小马），他为审阅译文花了大量的时间和精力，他的耐心和细致让我十分敬佩。感谢朱宁（白鸦）和周筠老师的引荐让我得以参与本书的翻译。还要感谢博文视点的编辑们在本书翻译过程中给予的极大理解和帮助。

我们在本书翻译过程中力求保持行文流畅，但纰漏在所难免，恳请广大读者批评指正。关于本书的任何意见或想法，欢迎发送邮件至 hpj.feedback@gmail.com。

最后，希望本书能帮助业界同仁打造出性能更为卓越的 Web 产品。

丁琛

前言
Preface

当 JavaScript 作为 Netscape Navigator 浏览器的一部分在 1996 年首次出现时，性能问题并不重要。当时的互联网仍处在发展初期，各个方面都很慢。从拨号上网到低配置的家用电脑，上网冲浪往往比任何事情都需要耐心。人们都做好了等待网页加载的心理准备，页面加载能完成就是一件值得庆祝的事了。

JavaScript 最初的目标是改善网页的用户体验。JavaScript 能代替服务器处理页面中类似表单验证的简单任务，这样做节省了与服务器连接的大量时间。想象一下当你填完一个很长的表单，提交后等待了 30~60 秒，却得到一个字段出错的信息是什么感觉。显而易见，JavaScript 为早期的互联网用户节省了很多时间。

互联网的发展
The Internet Evolves

在接下来的 10 年里，电脑和互联网不断发展。首先，两者都变得更快。高速的微处理器，内存的廉价供应，以及光纤连接的出现将互联网推向了一个新的时代。随着高速网络的普及，网页开始变得丰富，并且承载着更多的信息和多媒体内容。Web 从简单的关联文档演变成了各式各样的设计和界面。一切都变了，除了一样东西，那就是 JavaScript。

这项曾用于节省服务器消耗的技术日益普及，但代码也从数十行的发展到成百上千行。IE 4 和动态 HTML [译注1]（改变页面显示而无需重新加载的技术）的推出更是使得网页中的 JavaScript 代码量只增不减。

最近一次浏览器的重大更新是文档对象模型（DOM）的推出，这是一个被 IE 5、Netscape 6 及 Opera 所一致接受的动态HTML 接口。紧接着，JavaScript 被标准化，并推出了 ECMA-262

[译注1] Dynamic HTML（简称 DHTML）是一种通过结合 HTML/JavaScript/CSS 与 DOM，来创建动态网页内容的方法。它的历史发展和更多信息请参见 *http://en.wikipedia.org/wiki/Dynamic_HTML*。

第三版。随着所有浏览器都支持 DOM，同时（或多或少）提供了对相同版本 JavaScript 的支持，Web 应用平台诞生了。尽管有如此巨大的飞跃，有了编写 JavaScript 的通用 API，但是负责执行代码的 JavaScript 引擎几乎没有变化。

为什么优化是必要的

Why Optimization Is Necessary

在 1996 年，JavaScript 引擎只要能支持页面里数十行的 JavaScript 代码就好，而今天，却运行着成千上万行 JavaScript 代码的 Web 应用。从许多方面来说，如果不是因为浏览器自身在语言管理和基础设施方面的落后，JavaScript 本可能取得更大规模的成功。IE 6 就是一个明证，发布之初，它的稳定性和性能都被人们称颂，但后来却因为自身的 Bug 和反应迟钝而被痛批为令人讨厌的 Web 应用平台。

事实上，IE 6 并没有变慢，它只是被寄予了厚望。2001 年 IE 6 刚发布时出现的各类早期 Web 应用比 2005 年后出现的应用更轻量，JavaScript 代码也远没有那么多。JavaScript 代码数量的增长带来的影响变得明显，IE 6 的 JavaScript 引擎吃不消了，原因在于它的“静态垃圾回收机制”[译注2]。该引擎监视内存中固定数量的对象来确定何时进行垃圾回收。早期的 Web 应用开发人员很少会遇到这个极限值，随着更多的 JavaScript 代码产生越来越多的对象，复杂的 Web 应用开始频繁遭遇这个门槛。问题变得清晰起来：JavaScript 开发人员和 Web 应用都在发展，而 JavaScript 引擎却没有。

尽管其他浏览器有着更加完善的垃圾回收机制和更好的运行性能，但大多数仍然使用 JavaScript 解释器来执行代码。解释性代码天生就没有编译性代码快，因为解释性代码必须经历把代码转化成计算机指令的过程。无论解释器怎样优化和多么智能，它总是会带来一些性能损耗。

编译器已经有了各种各样的优化，使得开发人员可以按照他们想要的方式编写代码，而不需要担心是否是最优。编译器可以基于词法分析去判断代码想实现什么，然后产生出能完成任务的运行最快的机器码来进行优化。解释器很少有这样的优化，这很大程度上意味着，代码怎么写，就被怎么执行。

实际上，通常在其他语言中由编译器处理的优化，在 JavaScript 中却要求开发人员来完成。

[译注2]: “Static Garbage Collection”的详细解释请参见 *http://joses.st.ewi.tudelft.nl/static-gc.html*。关于 IE 6 的垃圾回收机制原理，请参见 MSDN 的文章 *http://blogs.msdn.com/ericlippert/archive/2003/09/17/53038.aspx*。

下一代 JavaScript 引擎

Next-Generation JavaScript Engines

2008 年，JavaScript 引擎迎来了第一次大的性能升级。Google 发布了全新的浏览器，名为 Chrome。Chrome 是第一款采用优化后的 JavaScript 引擎的浏览器，该引擎的研发代号为 V8。V8 是一款为 JavaScript 打造的实时（JIT）编译引擎，它把 JavaScript 代码转化为机器码来执行，所以给人的感觉是 JavaScript 的执行速度超快。

其他浏览器紧跟着也优化了它们的 JavaScript 引擎。Safari 4 发布了名为 SquirrelFish Extreme（或称为 Nitro）的 JIT JavaScript 引擎，而 Firefox 3.5 的 TraceMonkey 引擎对频繁执行的代码路径做了优化[译注3]。

这些全新的 JavaScript 引擎带来的是编译器层面的优化，这也是它应该做的。或许有一天，开发人员完全无须关心代码的性能优化。可是，那一天还未到来。

性能依然需要关注

Performance Is Still a Concern

尽管核心 JavaScript 的执行速度已经有所提高，但 JavaScript 仍然有多个方面的问题在新的引擎中没有被处理。网络延迟导致的滞缓和影响页面外观的操作尚未得到浏览器的充分优化。尽管诸如函数内联、代码合并以及字符串连接算法等简单的优化很容易通过编译器进行优化，但对动态且多层结构的 Web 应用程序来说，这些优化只能解决部分的性能问题。

然而全新的 JavaScript 引擎让我们似乎看到未来高速互联网的模样，在可预见到的未来，当今的性能话题仍然具有相关性和重要性。

本书中讨论的技术和方案涉及 JavaScript 的各个方面，内容涵盖运行时间、下载、DOM 操作、页面生存周期等。这些话题仅仅是一小部分，它们相关的核心（ECMAScript）性能可能随着 JavaScript 的不断进步而变得无关紧要，但这还有待时日。

其他的话题则针对那些更快的 JavaScript 引擎也力不能及的方面：DOM 交互、网络延迟、JavaScript 的阻塞和并发下载等。这些话题在未来依然重要，而且还会再受到更大关注，它们需要从底层研究 JavaScript 执行时间才能继续提高。

[译注3] 截止到译者翻译本书之时，Firefox 4.0 版本已经推出 beta 版本，并宣称启用了性能更优的 JaegerMonkey 引擎。关于浏览器脚本引擎的性能演化及更多其他信息，请参见 *http://zh.wikipedia.org/zh-cn/JavaScript* 引擎。

本书的组织
How This Book Is Organized

本书章节的组织方式基于一个正常的 JavaScript 开发周期。最开始的第 1 章讨论页面加载 JavaScript 的最佳方式。第 2 章到第 8 章专注于那些有助于你的 JavaScript 代码尽可能高效运行的编程技术。第 9 章讨论构建和部署 JavaScript 文件到生产环境的最佳方法，第 10 章提到的性能工具能帮助你找出代码部署后的潜在问题。以下 5 章是由特约作者完成的：

- 第 3 章　DOM 编程，由 Stoyan Stefanov 完成。
- 第 5 章　字符串和正则表达式，由 Steven Levithan 完成。
- 第 7 章　Ajax，由 Ross Harmes 完成。
- 第 9 章　构建并部署高性能 JavaScript 应用，由 Julien Lecomte 完成。
- 第 10 章　工具，由 Matt Sweeney 完成。

这些作者都是对 Web 开发社区有着重要贡献的资深 Web 开发人员。为便于你识别这些章节，他们的名字会出现在各自章节的起始页。

JavaScript 加载
JavaScript Loading

第 1 章，“加载和执行”，从 JavaScript 的基础开始：把代码加载到页面。要提高 JavaScript 的性能，首先要用最高效的方式加载代码到页面中。本章专注于与加载 JavaScript 代码相关的性能问题，并给出了几种方法以减轻它带来的负面影响。

编码技术
Coding Technique

JavaScript 中大量性能问题的来源是因为使用低效的算法或工具编写出的糟糕代码。接下来的 7 个章节专注于找出问题代码并给出更快替代方案来完成相同的任务。

第 2 章“数据访问”，重点介绍了 JavaScript 如何存取脚本里的数据。数据的储存位置同数据类型同样重要，本章解释了作用域链和原型链对脚本整体性能的影响。

Stoyan Stefanov，他对 Web 浏览器的内部工作机制十分在行，所以由他编写第 3 章“DOM 编程”。Stoyan 解释了在 JavaScript 中 DOM 交互比其他类型的操作要慢的原因是因为它的实现机制。他论述了 DOM 相关的所有内容，包括描述重绘和重排是如何拖慢你的代码的。

第 4 章“算法和流控制”，解释了常用编程模式（比如循环和迭代）是如何影响运行期性能的。本章还讨论了诸如 memoization 的优化技术以及浏览器的 JavaScript 运行期限制。

许多 Web 应用程序都会执行复杂的的 JavaScript 字符串操作，字符串专家 Steven Levithan 在第 5 章“字符串和正则表达式”全面介绍了相关主题。Web 开发人员已经与浏览器低效的字符串处理奋战多年，Steven 会解释为什么一些字符串操作起来会慢，以及如何应对。

第 6 章“快速响应的用户界面”，聚焦于用户体验。JavaScript 可能会导致浏览器假死，让用户感到无比沮丧。本章讨论了多种技术来确保用户界面总是处于可快速响应的状态。

第 7 章“Ajax”，Ross Harmes 讨论了实现客户端-服务端快速通信的最佳方法。Ross 还提到数据格式是如何影响 Ajax 性能的以及为什么 `XMLHttpRequest` 并非总是最佳选择。

第 8 章“编程实践”介绍了一些专门针对 JavaScript 的最佳实践。

部署

Deployment

JavaScript 代码一旦编写完成并通过测试，就是时候向用户发布了。然而，你不应当直接把源码放到生产环境。Julien Lecomte 在第 9 章“构建和部署高性能 JavaScript 应用”中介绍了如何使用一个构建系统去自动压缩文件，并使用 HTTP 压缩传输内容到浏览器。

测试

Testing

当所有 JavaScript 代码部署完成，下一步就要开始性能测试。Matt Sweeney 在第 10 章“工具”里的内容涵盖了测试方法论和相关工具。他介绍了如何使用 JavaScript 来衡量性能，还描述了两类通用工具，一类用于评估 JavaScript 运行期性能，另一类通过使用 HTTP 嗅探来发现隐藏的性能问题。

本书读者

Who This Book Is For

本书面向有中高级 JavaScript 经验且希望提升 Web 应用性能的 Web 开发人员。

本书的约定

Conventions Used in This Book

本书使用下列排版约定：

斜体（*Italic*）

表示专业词汇、链接（URL）、文件名和文件扩展名。

等宽字体（`Constant width`）

表示广义上的计算机代码，它们包括变量或函数名、数据库、数据类型、环境变量、语句和关键字。

这个图标表示提示、建议或一般说明。

这个图标表示警告或提醒。

代码用例

Using Code Examples

这本书是为了帮助你做好工作。一般来说，你可以在程序和文档中使用本书的代码。你无须联系我们获取许可。例如，使用来自本书的几段代码写一个程序是不需要许可的。出售和散布 O'Reilly 书中用例的光盘（CD-ROM）是需要许可的。通常引用本书用例和代码来回答问题是不需要许可的。把本书中大量的用例代码并入到你的产品文档中是需要许可的。

我们赞赏但不强求注明信息来源。一条信息来源通常包括标题、作者、出版者和国际标准书号（ISBN）。例如："High Performance JavaScript, by Nicholas C. Zakas. Copyright 2010 Yahoo!, Inc.,978-0-596-80279-0."。

如果你感到对示例代码的使用超出了正当引用或这里给出的许可范围，请随时通过 *permissions@oreilly.com* 联系我们。

意见和问题
Comments and Questions

请将对本书的评价和存在的问题通过如下地址告知出版者。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

对于本书的评论或技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

想了解关于 O'Reilly 图书、课程、会议和新闻的更多信息，请参阅我们的网站：

http://www.oreilly.com
http://www.oreilly.com.cn

与本书有关的在线信息如下所示：

http://www.oreilly.com/catalog/9780596802790（原书）

致谢

Acknowledgments

首先，也是最重要的，我要感谢本书的所有特约作者：Matt Sweeney、Stoyan Stefanov、Stephen Levithan、Ross Harmes 和 Julien Lecomte。他们的专业技能和知识的结合使得本书的编写过程让人兴奋，让本书更加值得期待。

感谢所有我有幸会面和交流的来自世界各地的性能专家，特别是 Steve Souders、Tenni Theurer 和 Nicole Sullivan。你们三位帮助我扩大了在 Web 性能方面的视野，我心存感激。

还要感谢每一位帮忙审阅本书的人们，包括 Ryan Grove、Oliver Hunt、Matthew Russell、Ted Roden、Remy Sharp 和 Venkateswaran Udayasankar。

还要特别感谢 O'Reilly 和 Yahoo! 的每一位成就此书的人。我在 2006 年刚刚加入 Yahoo! 时就希望为它写一本书，Yahoo! Press 让这件事成为现实。

目录

Table of contents

第 1 章　加载和执行 1

脚本位置 2

组织脚本 4

无阻塞的脚本 5

延迟的脚本 5

动态脚本元素 6

XMLHttpRequest 脚本注入 9

推荐的无阻塞模式 10

小结 14

第 2 章　数据存取 15

管理作用域 16

作用域链和标识符解析 16

标识符解析的性能 19

改变作用域链 21

动态作用域 24

闭包、作用域和内存 24

对象成员 27

原型 27

原型链 29

嵌套成员 30

缓存对象成员值 31

小结 33

第 3 章　DOM 编程 35

浏览器中的 DOM 35

天生就慢 36
DOM 访问与修改 36
innerHTML 对比 DOM 方法 37
节点克隆 41
HTML 集合 42
遍历 DOM 46
重绘与重排 50
重排何时发生 51
渲染树变化的排队与刷新 51
最小化重绘和重排 52
缓存布局信息 56
让元素脱离动画流 56
IE 和:hover 57
事件委托 57
小结 59

第 4 章　算法和流程控制 61

循环 61
循环的类型 61
循环性能 63
基于函数的迭代 67
条件语句 68
if-else 对比 switch 68
优化 if-else 70
查找表 72
递归 73
调用栈限制 74
递归模式 75
迭代 76
Memoization 77
小结 79

第 5 章　字符串和正则表达式 81

字符串连接 81
加（+）和加等（+=）操作符 82

数组项合并 84
String.prototype.concat 86
正则表达式优化 87
正则表达式工作原理 88
理解回溯 89
回溯失控 91
基准测试的说明 96
更多提高正则表达式效率的方法 96
何时不使用正则表达式 99
去除字符串首尾空白 99
使用正则表达式去首尾空白 99
不使用正则表达式去除字符串首尾空白 102
混合解决方案 103
小结 104

第 6 章　快速响应的用户界面 107

浏览器 UI 线程 107
浏览器限制 109
多久才算“太久” 110
使用定时器让出时间片段 111
定时器基础 112
定时器的精度 114
使用定时器处理数组 114
分割任务 116
记录代码运行时间 118
定时器与性能 119
Web Workers 120
Worker 运行环境 120
与 Worker 通信 121
加载外部文件 122
实际应用 122
小结 124

第 7 章　Ajax

数据传输 125

请求数据 125
发送数据 131
数据格式 134
XML 134
JSON 137
HTML 141
自定义格式 142
数据格式总结 144
Ajax 性能指南 145
缓存数据 145
了解 Ajax 类库的局限 148
小结 149

第8章 编程实践 151

避免双重求值（Double Evaluation） 151
使用 Object/Array 直接量 153
避免重复工作 154
延迟加载 154
条件预加载 156
使用速度快的部分 156
位操作 156
原生方法 159
小结 161

第9章 构建并部署高性能 JavaScript 应用 163

Apache Ant 163
合并多个 JavaScript 文件 165
预处理 JavaScript 文件 166
JavaScript 压缩 168
构建时处理对比运行时处理 170
JavaScript 的 HTTP 压缩 170
缓存 JavaScript 文件 171
处理缓存问题 172
使用内容分发网络（CDN） 173
部署 JavaScript 资源 173

敏捷 JavaScript 构建过程 174
小结 175

第 10 章　工具 177

JavaScript 性能分析 178
YUI Profiler 179
匿名函数 182
Firebug 183
    控制台面板分析工具 183
    Console API 184
    网络面板 185
IE 开发人员工具 186
Safari Web 检查器（Web Inspector） 188
    分析面板 189
    资源面板 191
Chrome 开发人员工具 192
脚本阻塞 193
Page Speed 194
Fiddler 196
YSlow 198
dynaTrace Ajax Edition 199
小结 202

索引 203

第 1 章

加载和执行

Loading and Execution

JavaScript 在浏览器中的性能，可以认为是开发者所面临的最严重的可用性问题。这个问题因 JavaScript 的阻塞特性变得复杂，也就是说当浏览器在执行 JavaScript 代码时，不能同时做其他任何事情。事实上，多数浏览器使用单一进程来处理用户界面（UI）刷新和 JavaScript 脚本执行，所以同一时刻只能做一件事。JavaScript 执行过程耗时越久，浏览器等待响应的时间就越长。

简单说，这意味着`<script>`标签每次出现都霸道地让页面等待脚本的解析和执行。无论当前的 JavaScript 代码是内嵌的还是包含在外链文件中，页面的下载和渲染都必须停下来等待脚本执行完成。这是页面生存周期中的必要环节，因为脚本执行过程中可能会修改页面内容。一个典型的例子就是在页面中使用 `document.write()`（经常用来显示广告）。例如：

```
<html>
<head>
   <title>Script Example</title>
</head>
<body>
  <p>
  <script  type="text/javascript">
    document.write("The  date  is " + (new  Date()).toDateString());
  </script>
  </p>
</body>
</html>
```

当浏览器遇到`<script>`标签时，当前 HTML 页面无从获知 JavaScript 是否会向`<p>`标签添加内容，或引入其他元素，或甚至关闭该标签。因此，这时浏览器会停止处理页面，先执行 JavaScript 代码，然后再继续解析和渲染页面。同样的情况也发生在使用 `src` 属性加载

JavaScript 的过程中，浏览器必须先花时间下载外链文件中的代码，然后解析并执行。在这个过程中，页面渲染和用户交互是完全被阻塞的。

提示：本章内容大量参考了雅虎特别性能小组（*http://developer.yahoo.com/performance/*）和《高性能网站建设》（O'Reilly）及《高性能网站建设进阶指南》（O'Reilly）的作者 Steve Souders 在 JavaScript 影响页面下载性能方面的研究结果。

脚本位置

Script Positioning

HTML4 规范指出`<script>`标签可以放在 HTML 文档的`<head>`或`<body>`中，并允许出现多次。按照惯例，`<script>`标签用来加载出现在`<head>`中的外链 JavaScript 文件，挨着的`<link>`标签用来加载外部 CSS 文件或者其他页面元信息。理论上来说，把与样式和行为有关的脚本放在一起，并先加载它们，这样做有助于确保页面渲染和交互的正确性。例如：

```
<html>
<head>
   <title>Script Example</title>
   <--  Example of inefficient script positioning -->
   <script type="text/javascript" src="file1.js"></script>
   <script type="text/javascript" src="file2.js"></script>
   <script type="text/javascript" src="file3.js"></script>
   <link rel="stylesheet" type="text/css"  href="styles.css">
</head>
<body>
  <p>Hello world!</p>
</body>
</html>
```

这些看似正常的代码实际上有十分严重的性能问题：在`<head>`中加载了三个 JavaScript 文件。由于脚本会阻塞页面渲染，直到它们全部下载并执行完成后，页面的渲染才会继续。因此页面的性能问题会很明显。请记住，浏览器在解析到`<body>`标签之前，不会渲染页面的任何部分。把脚本放到页面顶部将会导致明显的延迟，通常表现为显示空白页面，用户无法浏览内容，也无法与页面进行交互。瀑布图可以帮助我们更清晰地理解性能问题发生的原因，瀑布图描述了每个资源文件的下载过程。图 1-1 显示了页面加载过程中脚本和样式文件的下载过程。

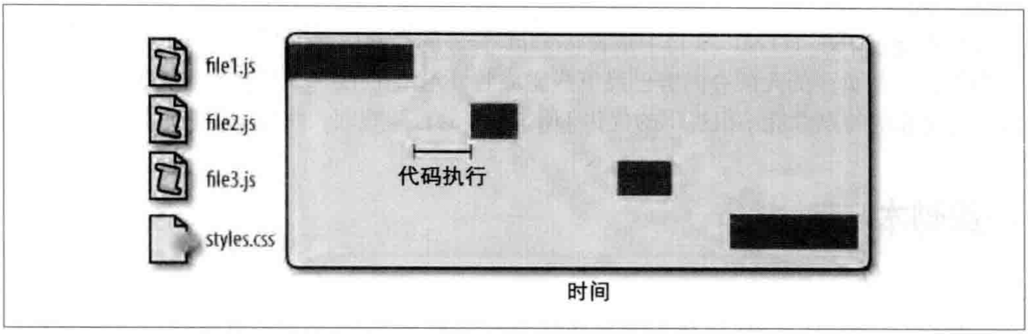


图 1-1　执行 JavaScript 代码阻塞其他文件的下载

图 1-1 显示了一个有趣的情况。第一个 JavaScript 文件开始下载，与此同时阻塞了页面其他文件下载。此外，从 file1.js 下载完成到 file2.js 开始下载前存在一个延时，这段时间正好是 file1.js 的执行过程。每个文件必须等到前一个文件下载并执行完成才会开始下载。在这些文件逐个下载过程中，用户看到的是一片空白。这是当今大多数浏览器的行为特征。

IE 8、Firefox 3.5、Safari 4 和 Chrome 2 都允许并行下载 JavaScript 文件。这是个好消息，因为`<script>`标签在下载外部资源时不会阻塞其他`<script>`标签。遗憾的是，JavaScript 下载过程仍然会阻塞其他资源的下载，比如图片。尽管脚本的下载过程不会互相影响，但页面仍然必须等待所有 JavaScript 代码下载并执行完成才能继续。因此，尽管最新的浏览器通过允许并行下载提高了性能，但问题尚未完全解决。脚本阻塞仍然是一个问题。

由于脚本会阻塞页面其他资源的下载，因此推荐将所有的`<script>`标签尽可能放到 `<body>` 标签的底部，以尽量减少对整个页面下载的影响。例如：

```
<html>
<head>
   <title>Script Example</title>
   <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <p>Hello world!</p>

   <-- Example of recommended script positioning -->
   <script type="text/javascript" src="file1.js"></script>
   <script type="text/javascript" src="file2.js"></script>
   <script type="text/javascript" src="file3.js"></script>
</body>
</html>
```

这段代码展示了在 HTML 文档中放置`<script>`标签的推荐位置。尽管脚本下载会阻塞另一个脚本，但是页面的大部分内容已经下载完成并显示给了用户，因此页面下载不会显得太慢。这是雅虎特别性能小组提出的优化 JavaScript 的首要规则：将脚本放在底部。

组织脚本

Grouping Scripts

由于每个`<script>`标签初始下载时都会阻塞页面渲染，所以减少页面包含的`<script>`标签数量有助于改善这一情况。这不仅仅针对外链脚本，内嵌脚本的数量同样也要限制。浏览器在解析 HTML 页面的过程中每遇到一个`<script>`标签，都会因执行脚本而导致一定的延时，因此最小化延迟时间将会明显改善页面的总体性能。

提示：Steve Souders 还发现，把一段内嵌脚本放在引用外链样式表的`<link>`标签之后会导致页面阻塞去等待样式表的下载。这样做是为了确保内嵌脚本在执行时能获得最精准的样式信息。因此，Souders 建议永远不要把内嵌脚本紧跟在`<link>`标签后面。

这个问题在处理外链 JavaScript 文件时略有不同。考虑到 HTTP 请求会带来额外的性能开销，因此下载单个 100KB 的文件将比下载 4 个 25KB 的文件更快。也就是说，减少页面中外链脚本文件的数量将会改善性能。

通常一个大型网站或网络应用需要依赖数个 JavaScript 文件。你可以把多个文件合并成一个，这样只需要引用一个`<script>`标签，就可以减少性能消耗。文件合并的工作可通过离线的打包工具（第 9 章会讨论到）或者类似 Yahoo! combo handler 的实时在线服务来实现。

雅虎提供了合并处理器，以供通过他们的内容传输网络（CDN）来分发 Yahoo! User Interface（YUI）Library 文件时使用。任何网站都可以使用一个把指定文件合并处理后的 URL 来获取任意数量的 YUI 文件。例如，下面的 URL 包含了两个文件：*http://yui.yahooapis.com/combo?2.7.0/build/yahoo/yahoo-min.js&2.7.0/build/event/event-min.js*。

这个 URL 加载了 2.7.0 版本的 yahoo-min.js 和 event-min.js 文件。这两个文件在服务器上是独立存在的，但通过请求以上网址它们会被合并。原来需要用两个`<script>`标签分别加载两个文件，而现在用一个`<script>`标签即可同时加载：

```
<html>
<head>
   <title>Script Example</title>
   <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <p>Hello world!</p>

   <-- 推荐的脚本存放位置 -->
   <script type="text/javascript" src="
http://yui.yahooapis.com/combo?2.7.0/build/yahoo/yahoo-min.js&
2.7.0/build/event/event-min.js"></script>
</body>
</html>
```

这段代码只有一个位于页面底部的`<script>`标签，它却加载了多个 JavaScript 文件。这是在 HTML 页面中引入多个外链 JavaScript 文件的最佳实践。

无阻塞的脚本

Nonblocking Scripts

JavaScript 倾向于阻止浏览器的某些处理过程，如 HTTP 请求和用户界面更新，这是开发者所面临的最显著的性能问题。减少 JavaScript 文件大小并限制 HTTP 请求数仅仅是创建响应迅速的 Web 应用的第一步。Web 应用的功能越丰富，所需要的 JavaScript 代码就越多，所以精简源代码并不总是可行。尽管下载单个较大的 JavaScript 文件只产生一次 HTTP 请求，却会锁死浏览器一大段时间。为避免这种情况，你需要向页面中逐步加载 JavaScript 文件，这样做在某种程度上来说不会阻塞浏览器。

无阻塞脚本的秘诀在于，在页面加载完成后才加载 JavaScript 代码。用专业术语来说，这意味着在 `window` 对象的 `load` 事件触发后再下载脚本。有多种方式可以实现这一效果。[译注1]

延迟的脚本

Deferred Scripts

HTML 4 为`<script>`标签定义了一个扩展属性：`defer`。`Defer` 属性指明本元素所含的脚本不会修改 DOM，因此代码能安全地延迟执行。该属性只有 IE 4+和 Firefox 3.5+的浏览器支持，所以它不是一个理想的跨浏览器解决方案。[译注2]在其他浏览器中，`defer` 属性会被直接忽略，因此`<script>`标签会以默认的方式处理（即会造成阻塞）。然而，如果你的目标浏览器支持的话，这仍然是个有用的解决方案。下面是一个例子：

```
<script type="text/javascript" src="file1.js" defer></script>
```

[译注1] 除下个小节提到的 defer 属性外，HTML5 规范中引入了 async 属性，用于异步加载脚本。async 与 defer 的相同点是采用并行下载，在下载过程中不会产生阻塞。区别在于执行时机，async 是加载完成后自动执行，而 defer 需要等待页面完成后执行。详见：http://www.w3.org/TR/html5/single-page.html#attr-script-defer

[译注2] defer：标签目前已被所有主流浏览器支持，详细版本列表详见：http://caniuse.com/#feat=script-defer

带有 `defer` 属性的`<script>`标签可以放置在文档的任何位置。对应的 JavaScript 文件将在页面解析到`<script>`标签时开始下载，但并不会执行，直到 DOM 加载完成（`onload` 事件被触发前）。当一个带有 `defer` 属性的 JavaScript 文件下载时，它不会阻塞浏览器的其他进程，因此这类文件可以与页面中的其他资源并行下载。

任何带有 `defer` 属性的`<script>`元素在 DOM 完成加载之前都不会被执行，无论内嵌或外链脚本都是如此。下面的例子展示了 `defer` 属性如何影响脚本行为[译注3]：

```
<html>
<head>
    <title>Script Defer Example</title>
</head>
<body>
    <script defer>
        alert("defer");
    </script>
    <script>
        alert("script");
    </script>
    <script>
        window.onload = function(){
            alert("load");
        };
    </script>
</body>
</html>
```

这段代码在页面处理过程中弹出三次提示框。不支持 `defer` 属性的浏览器的弹出顺序是“defer”、“script”、“load”。而在支持 `defer` 属性的浏览器上，弹出的顺序是：“script”、“defer”、“load”。请注意，带有 defer 属性的`<script>`元素不是跟在第二个后面执行，而是在 `onload` 事件处理器执行之前被调用。

动态脚本元素

Dynamic Script Elements

由于文档对象模型（DOM）的存在，你可以用 JavaScript 动态创建 HTML 中的几乎所有内容。其原因在于，`<script>`元素与页面中其他元素并无差异：都能通过 DOM 进行引用，都能在文档中移动、删除，或是被创建。用标准的 DOM 方法可以很容易地创建一个新的`<script>`元素：

[译注3] 考虑到原书发布的时效性，以下代码仅适用于低版本 IE 浏览器。如今所有主流浏览器都已实现对 defer 属性的支持，根据 W3C 的 HTML5 规范定义：defer 属性仅当 src 属性声明时才生效。请参考链接：http://www.w3.org/TR/html5/single-page.html#attr-script-defer

```
var  script = document.createElement("script");
script.type = "text/javascript";
script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);[译注4]
```

这个新创建的`<script>`元素加载了 file1.js 文件。文件在该元素被添加到页面时开始下载。这种技术的重点在于：无论在何时启动下载，文件的下载和执行过程不会阻塞页面其他进程。你甚至可以将代码放到页面`<head>`区域而不会影响页面其他部分（用于下载该文件的 HTTP 链接本身的影响除外）。

> **提示**：通常来讲，把新创建的`<script>`标签添加到`<head>`标签里比添加到`<body>`里更保险，尤其是在页面加载过程中执行代码时更是如此。当`<body>`中的内容没有全部加载完成时，IE 可能会抛出一个“操作已中止”的错误信息[译注5]。

使用动态脚本节点下载文件时，返回的代码通常会立刻执行（除了 Firefox 和 Opera，它们会等待此前所有动态脚本节点执行完毕）。当脚本“自执行”时，这种机制运行正常。但是当代码只包含供页面其他脚本调用的接口时，就会有问题。在这种情况下，你必须跟踪并确保脚本下载完成且准备就绪。这可以用动态`<script>`节点触发的事件来实现。

Firefox, Opera, Chrome 和 Safari 3 以上版本会在`<script>`元素接收完成时触发一个 `load` 事件。因此，你可以通过侦听此事件来获得脚本加载完成时的状态：

```
var script = document.createElement("script")
script.type = "text/javascript";

//Firefox, Opera, Chrome, Safari 3+
script.onload = function(){
    alert("Script loaded!");
};

script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

IE 支持另一种实现方式，它会触发一个 `readystatechange` 事件。`<script>`元素提供一个 `readyState` 属性，它的值在外链文件的下载过程的不同阶段会发生变化，该属性有五种取值：

[译注4] 在支持 HTML5 的浏览器中，可直接使用 document.head 获得<head>元素的引用。

[译注5] 《可怕的中止操作错误》一文对该问题有更深入的讨论：*http://www.nczonline.net/blog/2008/03/17/the-dreaded-operation-aborted-error*。

`"uninitialized"`
  初始状态

`"loading"`
  开始下载

`"loaded"`
  下载完成

`"interactive"`
  数据完成下载但尚不可用

`"complete"`
  所有数据已准备就绪

微软的相关文档表明，在`<script>`元素生命周期中，并非`readyState`的每个取值都会被用到，该文档也没有指出哪些一定会被用到。实际应用中，最有用的两个状态是`"loaded"`和`"complete"`。IE在标识最终状态的`readyState`的值时并不一致，有时`<script>`元素到达`"loaded"`状态而从不会到达`"complete"`，有时甚至不经过`"loaded"`就到达`"complete"`状态。使用`readystatechange`事件最靠谱的方式是同时检查这两种状态，只要其中任何一个触发，就删除事件处理器（以确保事件不会处理两次）。

```
var  script = document.createElement("script")
script.type = "text/javascript";

//IE
script.onreadystatechange = function(){
    if (script.readyState == "loaded" || script.readyState == "complete"){
         script.onreadystatechange = null;
         alert("Script loaded.");
    }
};

script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

在大多数情况下，你需要使用一个单一的方法来动态加载JavaScript文件，下面的函数封装了标准及IE特有的实现方法：

```
function loadScript(url, callback){

     var script = document.createElement("script")
     script.type = "text/javascript";

     if  (script.readyState){ //IE
          script.onreadystatechange = function(){
              if (script.readyState == "loaded" || script.readyState == "complete"){
                  script.onreadystatechange = null;
                  callback();
              }
          };
```

```
    } else{ // 其他浏览器
        script.onload = function(){
            callback();
        };
    }

    script.src = url;
    document.getElementsByTagName("head")[0].appendChild(script);
}
```

这个函数接受两个参数：JavaScript 文件的 URL 和完成加载后的回调函数。函数中使用了特征检测（Feature detection）来决定在脚本处理过程中监听哪个事件。最后一步是给 `src` 属性赋值，然后将`<script>`元素添加到页面。`loadScript()`函数用法如下：

```
loadScript("file1.js", function(){
    alert("File is loaded!");
});
```

如果需要的话，你可以动态加载尽可能多的 JavaScript 文件到页面上，但一定要考虑清楚文件的加载顺序。在所有主流浏览器中，只有 Firefox 和 Opera 能保证脚本会按照你指定的顺序执行，其他浏览器将会按照从服务端返回的顺序下载和执行代码。你可以将下载操作串联起来以确保下载顺序，比如：

```
loadScript("file1.js", function(){
    loadScript("file2.js", function(){
        loadScript("file3.js", function(){
            alert("All files are loaded!");
        });
    });
});
```

这段代码会先加载 file1.js，等待 file1.js 加载完成后再加载 file2.js，以此类推最后加载 file3.js。尽管方案可行，但如果需要下载的文件较多，这个方案会带来一点管理上的麻烦。

如果多个文件的下载顺序很重要，更好的做法是把它们按正确顺序合并成一个文件。下载这个文件就能一次获得所有代码（由于这个过程是异步的，因此文件大一点不会有影响）。

动态脚本加载凭借着它在跨浏览器兼容性和易用的优势，成为最通用的无阻塞加载解决方案。

XMLHttpRequest 脚本注入

XMLHttpRequest Script Injection

另一种无阻塞加载脚本的方法是使用 XMLHttpRequest（XHR）对象获取脚本并注入页面中。

此技术会先创建一个 XHR 对象，然后用它下载 JavaScript 文件，最后通过创建动态`<script>`元素将代码注入页面中。下面是一个简单的例子：

```
var xhr = new XMLHttpRequest();
xhr.open("get", "file1.js", true);
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if (xhr.status  >= 200 && xhr.status < 300 || xhr.status == 304){
            var script = document.createElement("script");
            script.type = "text/javascript";
            script.text = xhr.responseText;
            document.body.appendChild(script);
        }

    }
};
xhr.send(null);
```

这段代码发送一个 GET 请求获取 file1.js 文件。事件处理函数 `onReadyStateChange` 检查 `readyState` 是否为 `4`，同时校验 HTTP 状态码是否有效（2XX 表示有效响应，304 意味着是从缓存读取）。如果收到了有效响应，就会创建一个`<script>`元素，设置该元素的 `text` 属性为从服务器接收到的 `responseText`。这样实际上相当于创建一个带有内联脚本的`<script>`标签。一旦新创建的`<script>`元素被添加到页面，代码就会立刻执行然后准备就绪。

这种方法的主要优点是，你可以下载 JavaScript 代码但不立即执行。由于代码是在`<script>`标签之外返回的，因此它下载后不会自动执行，这使得你可以把脚本的执行推迟到你准备好的时候。另一个优点是，同样的代码在所有主流浏览器中无一例外都能正常工作。

这种方法的主要局限性是 JavaScript 文件必须与所请求的页面处于相同的域，这意味着 JavaScript 文件不能从 CDN 下载。因此，大型的 Web 应用通常不会采用 XHR 脚本注入技术。

推荐的无阻塞模式

Recommended Nonblocking Pattern

向页面中添加大量 JavaScript 的推荐做法只需两步：先添加动态加载所需的代码，然后加载初始化页面所需的剩下的代码。因为第一部分的代码尽量精简，甚至可能只包含 `loadScript()` 函数，它下载执行都很快，所以不会对页面有太多影响。一旦初始代码就位，就用它来加载剩余的 JavaScript。例如：

```
<script type="text/javascript" src="loader.js"></script>
<script type="text/javascript">
    loadScript("the-rest.js", function(){
        Application.init();
    });
</script>
```

把这段加载代码放到`</body>`闭合标签之前。这样做有几个好处：首先，如前文提到，这样做确保了 JavaScript 执行过程中不会阻碍页面其他内容的显示。其次，当第二个 JavaScript 文件完成下载时，应用所需的所有 DOM 结构已经创建完毕，并做好了交互的准备，从而避免了需要另一个事件（比如`window.onload`）来检测页面是否准备好。

另一种方法是把`loadScript()`函数直接嵌入页面，从而避免多产生一次 HTTP 请求。例如：

```
<script type="text/javascript">
    function loadScript(url, callback){

        var  script = document.createElement("script")
        script.type = "text/javascript";

        if (script.readyState){ //IE
            script.onreadystatechange = function(){
                if  (script.readyState == "loaded" ||
                         script.readyState == "complete"){
                     script.onreadystatechange = null;
                     callback();
                }
            };
        }  else { // 其他浏览器
            script.onload = function(){
                callback();
            };
        }

        script.src = url;
        document.getElementsByTagName("head")[0].appendChild(script);
    }

    loadScript("the-rest.js", function(){
        Application.init();
    });
</script>
```

如果你决定采取后一种做法，建议你使用 YUI Compressor（见第 9 章）把初始化代码压缩到最小尺寸。

一旦页面初始化所需的代码完成下载，你可以继续自由地使用`loadScript()`去加载页面其他的功能所需的脚本。

YUI3 的方式

YUI3 有一个核心设计理念是：由页面中的少量代码来加载丰富的功能组件。在页面中使用 YUI3，只须引入如下 YUI 种子文件：

```
<script  type="text/javascript"
src="http://yui.yahooapis.com/combo?3.0.0/build/yui/yui-min.js"></script>
```

这个种子文件大小在 10KB 左右（Gzip 压缩后只有 6KB），但它包含的功能足以从雅虎 CDN 下载任意其他的 YUI 组件。比如，如果你要使用 DOM 工具包，只需在 YUI 的 `use()`方法中声明（`"dom"`）并且提供回调函数即可：

```
YUI().use("dom", function(Y){
    Y.DOM.addClass(docment.body, "loaded");
});
```

以上代码创建了一个 YUI 对象的新实例，并调用它的 `use()`方法。种子文件包含了文件名及依赖关系的所有信息，因此指定`"dom"`实际上会拼装出一个带有所有依赖文件组合的 URL，然后由动态脚本元素加载进来并执行。当所有代码都准备完成，回调方法会被执行，并接收当前 YUI 实例作为参数，于是你就可以立即使用新下载的功能。

LazyLoad 类库

Yahoo! Search 的工程师 Ryan Grove 创建了一个更为通用的延迟加载工具：LazyLoad（源代码：*http://github.com/rgrove/lazyload/*）。LazyLoad 是 `loadScript()`函数的增强版。该文件压缩后约 1.5KB（未使用 Gzip 压缩）。用法示例：

```
<script type="text/javascript" src="lazyload-min.js"></script>
<script type="text/javascript">
    LazyLoad.js("the-rest.js", function(){
        Application.init();
    });
</script>
```

LazyLoad 同样支持下载多个 JavaScript 文件，并能保证在所有浏览器中都以正确的顺序执行。要加载多个 JavaScript 文件，只需给 `LazyLoad.js()`方法传入一个 URL 数组：

```
<script type="text/javascript" src="lazyload-min.js"></script>
<script type="text/javascript">
    LazyLoad.js(["first-file.js", "the-rest.js"], function(){
        Application.init();
    });
</script>
```

尽管使用这种无阻塞的方式可以动态加载很多文件，但是建议尽量减少文件数。因为每次下载仍然是一个独立的 HTTP 请求，而且回调函数会等待所有文件都下载完成后才会执行。

提示：LazyLoad 同样可以动态加载 CSS 文件。这没有太大的意义，因为 CSS 文件本已是并行下载，不会阻塞页面的其他进程。

LABjs

另一个开源的无阻塞脚本加载工具是 Kyle Simpson 受 Steve Souders 的启发而编写的 LABjs（*http://labjs.com*）。该工具提供了对加载过程更精细的控制，并试图同时下载尽可能多的代码。LABjs 同样非常精简，压缩后只有 4.5KB（非 Gzip 压缩），因此有着最少的调用代码。用法举例：

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
    $LAB.script("the-rest.js")
        .wait(function(){
            Application.init();
        });
</script>
```

`$LAB.script()`方法用来定义需要下载的 JavaScript 文件，`$LAB.wait()`用来指定文件下载并执行完毕后所调用的函数。LABjs 鼓励链式操作，因此每个方法都会返回一个`$LAB`对象的引用。要下载多个 JavaScript 文件，只需链式调用另一个`$LAB.script()`方法：

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
    $LAB.script("first-file.js")
        .script("the-rest.js")
        .wait(function(){
            Application.init();
        });
</script>
```

LABjs 与众不同的是它管理依赖关系的能力。通常来说，连续的`<script>`标签意味着文件逐个下载（或串行，或并行，如前文所述）并按顺序执行。在某些情况下这样做有必要，但有时未必。

LABjs 允许你使用 `wait()`方法来指定哪些文件需要等待其他文件。在前面的例子中，first-file.js 的代码不能保证会在 the-rest.js 的代码前执行。为了确保这一点，你必须在第一个`script()`方法后调用 `wait()`：

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
    $LAB.script("first-file.js").wait()
        .script("the-rest.js")
        .wait(function(){
            Application.init();
        });
</script>
```

这时，尽管是并行下载，但 first-file.js 中的代码肯定能在 the-rest.js 前面执行。

小结
Summary

管理浏览器中的 JavaScript 代码是个棘手的问题，因为代码执行过程会阻塞浏览器的其他进程，比如用户界面绘制。每次遇到`<script>`标签，页面都必须停下来等待代码下载（如果是外链文件）并执行，然后继续处理其他部分。尽管如此，还是有几种方法能减少 JavaScript 对性能的影响：

- `</body>`闭合标签之前，将所有的`<script>`标签放到页面底部。这能确保在脚本执行前页面已经完成了渲染。
- 合并脚本。页面中的`<script>`标签越少，加载也就越快，响应也更迅速。无论外链文件还是内嵌脚本都是如此。
- 有多种无阻塞下载 JavaScript 的方法：
  - 使用`<script>`标签的 `defer` 属性；
  - 使用动态创建的`<script>`元素来下载并执行代码；
  - 使用 XHR 对象下载 JavaScript 代码并注入页面中。

通过以上策略，可以极大提高那些需要使用大量 JavaScript 的 Web 应用的实际性能。

第 2 章

数据存取

Data Access

计算机科学中有一个经典问题是通过改变数据的存储位置来获得最佳的读写性能，数据存储的位置关系到代码执行过程中数据的检索速度。在 JavaScript 中，这个问题相对简单，因为只有几种存储方案可供选择。不过，和其他编程语言一样，数据的存储位置会很大程度上影响其读取速度。JavaScript 中有下面四种基本的数据存取位置。

字面量

字面量只代表自身，不存储在特定位置。JavaScript 中的字面量有：字符串、数字、布尔值、对象、数组、函数、正则表达式，以及特殊的 null 和 undefined 值。

本地变量

开发人员使用关键字 var 定义的数据存储单元。

数组元素

存储在 JavaScript 数组对象内部，以数字作为索引。

对象成员

存储在 JavaScript 对象内部，以字符串作为索引。

每一种数据存储的位置都有不同的读写消耗。大多数情况下，从一个字面量和一个局部变量中存取数据的性能差异是微不足道的。访问数组元素和对象成员的代价则高一些，至于具体高出多少，很大程度上取决于浏览器。图 2-1 显示了在不同浏览器中，分别对这四种数据存储位置进行 200 000 次操作所用的时间。

老版本的浏览器使用传统的 JavaScript 引擎，比如 Firefox 3，IE 和 Safari 3.2，它们存取数据比优化后的 JavaScript 引擎会消耗更多的时间。总的来说，字面量和局部变量的访问速度快于数组项和对象成员的访问速度，只有 Firefox 3 是个例外，它优化了数组项的存取，所以

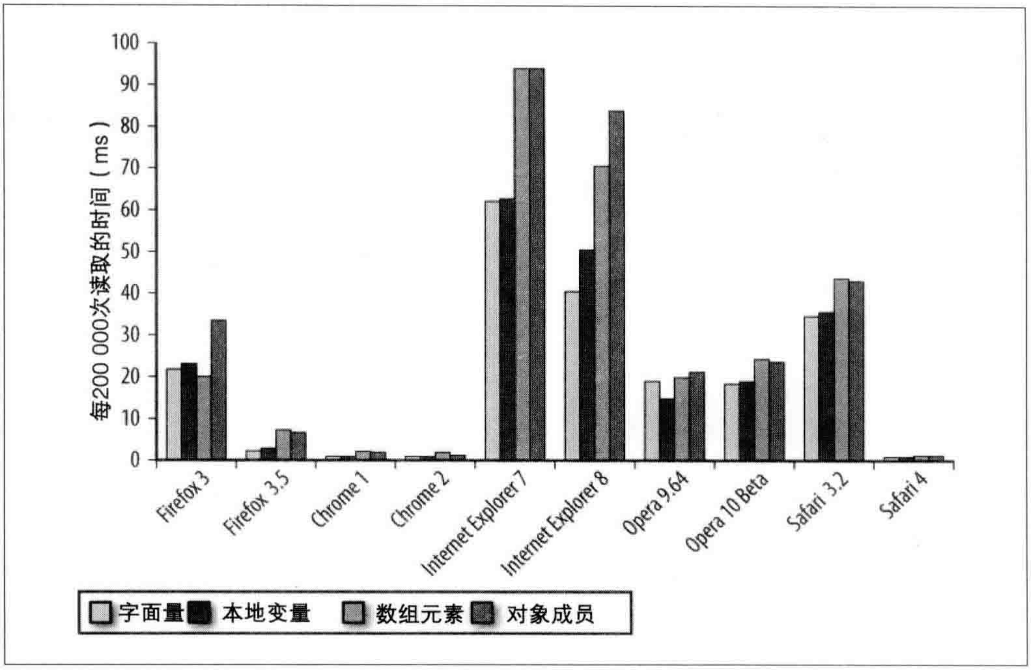


图 2-1 每 200 000 次读取变量存储位置所消耗的时间

速度快很多。尽管如此，通常的建议是，如果在乎运行速度，那么尽量使用字面量和局部变量，减少数组项和对象成员的使用。为此，有几种模式来定位和规避问题，以及优化代码。

管理作用域

Managing Scope

作用域概念是理解 JavaScript 的关键所在，不仅仅从性能角度，还包括从功能的角度。作用域对 JavaScript 有许多影响，从确定哪些变量可以被函数访问，到确定 this 的赋值。JavaScript 作用域同样关系到性能，要理解速度和作用域的关系，首先要正确地理解作用域的工作原理。

作用域链和标识符解析

Scope Chains and Identifier Resolution

每一个 JavaScript 函数都表示为一个对象，更确切地说，是 `Function` 对象的一个实例。Function 对象同其他对象一样，拥有可以编程访问的属性，和一系列不能通过代码访问而

仅供 JavaScript 引擎存取的内部属性。其中一个内部属性是`[[Scope]]`，由 ECMA-262 标准第三版（*http://www.ecma-international.org/publications/standards/Ecma-262.htm*）定义。

内部属性`[[Scope]]`包含了一个函数被创建的作用域中对象的集合。这个集合被称为函数的作用域链，它决定哪些数据能被函数访问。函数作用域中的每个对象被称为一个可变对象，每个可变对象都以“键值对”的形式存在。当一个函数创建后，它的作用域链会被创建此函数的作用域中可访问的数据对象所填充。例如，思考下面的全局函数：

```
function add(num1, num2){
    var sum = num1 + num2;
    return sum;
}
```

当函数 `add()`创建时，它的作用域链中插入了一个对象变量，这个全局对象代表着所有在全局范围内定义的变量。该全局对象包含诸如 `window`、`navigator` 和 `document` 等。图 2-2 说明了它们之间的关系（注意：图中只列举出全部变量中很少的一部分，此外还有很多）。

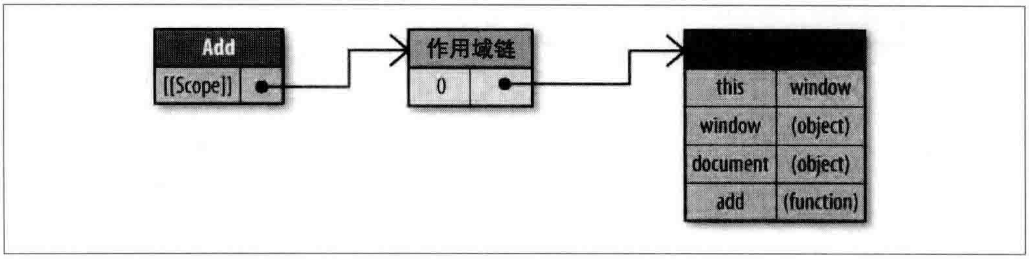


图 2-2　函数 add()的作用域链

函数 `add` 的作用域将会在执行时用到。假设执行如下代码：

```
var total = add(5, 10);
```

执行此函数时会创建一个称为执行环境（execution context）[译注1]的内部对象。一个执行环境定义了一个函数执行时的环境。函数每次执行时对应的执行环境都是独一无二的，所以多次调用同一个函数就会导致创建多个执行环境。当函数执行完毕，执行环境就被销毁。

[译注1] 另一种常见翻译是“执行上下文”。

每个执行环境都有自己的作用域链，用于解析标识符。当执行环境被创建时，它的作用域链初始化为当前运行函数的[[Scope]]属性中的对象。这些值按照它们出现在函数中的顺序，被复制到执行环境的作用域链中。这个过程一旦完成，一个被称为“活动对象（activation object）”的新对象就为执行环境创建好了。活动对象作为函数运行时的变量对象，包含了所有局部变量，命名参数，参数集合以及 `this`。然后此对象被推入作用域链的最前端。当执行环境被销毁，活动对象也随之销毁。图 2-3 显示了前面示例代码对应的执行环境和作用域链。

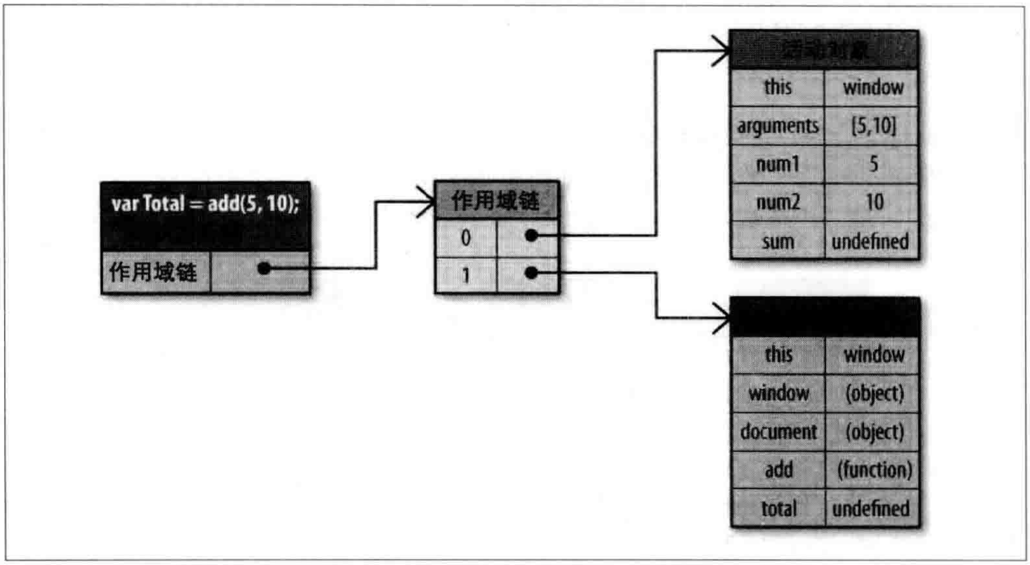

var Total = add(5, 10);
作用域链
作用域链
0
1
活动对象
this
window
arguments
[5,10]
num1
5
num2
10
sum
undefined
this
window
window
(object)
document
(object)
add
(function)
total
undefined

图 2-3　函数 add()执行期的作用域链

在函数执行过程中，每遇到一个变量，都会经历一次标识符解析过程以决定从哪里获取或存储数据。该过程搜索执行环境的作用域链，查找同名的标识符。搜索过程从作用域链头部开始，也就是当前运行函数的活动对象。如果找到，就使用这个标示符对应的变量；如果没找到，继续搜索作用域链中的下一个对象。搜索过程会持续进行，直到找到标识符，若无法搜索到匹配的对象，那么标识符将被视为是未定义的。在函数执行过程中，每个标识符都要经历这样的搜索过程。在前面的代码示例中，函数访问 `sum`、`num1`、`num2` 时都会产生搜索过程，正是这个搜索过程影响了性能。

提示：如果名字相同的两个变量存在于作用域链的不同部分，那么标识符就是遍历作用域链时最先找到的那个，也可以说，第一个变量遮蔽（Shadow）了第二个。

标识符解析的性能

Identifier Resolution Performance

标识符解析是有代价的，事实上没有哪种计算机操作可以不产生性能开销。在执行环境的作用域链中，一个标识符所在的位置越深，它的读写速度也就越慢。因此，函数中读写局部变量总是最快的，而读写全局变量通常是最慢的（优化 JavaScript 引擎在某些情况下能有所改善）。请记住，全局变量总是存在于执行环境作用域链的最末端，因此它也是最远的。图 2-4 和 2-5 显示了作用域链中不同深度标识符的解析速度。深度为 1 表示是局部变量。

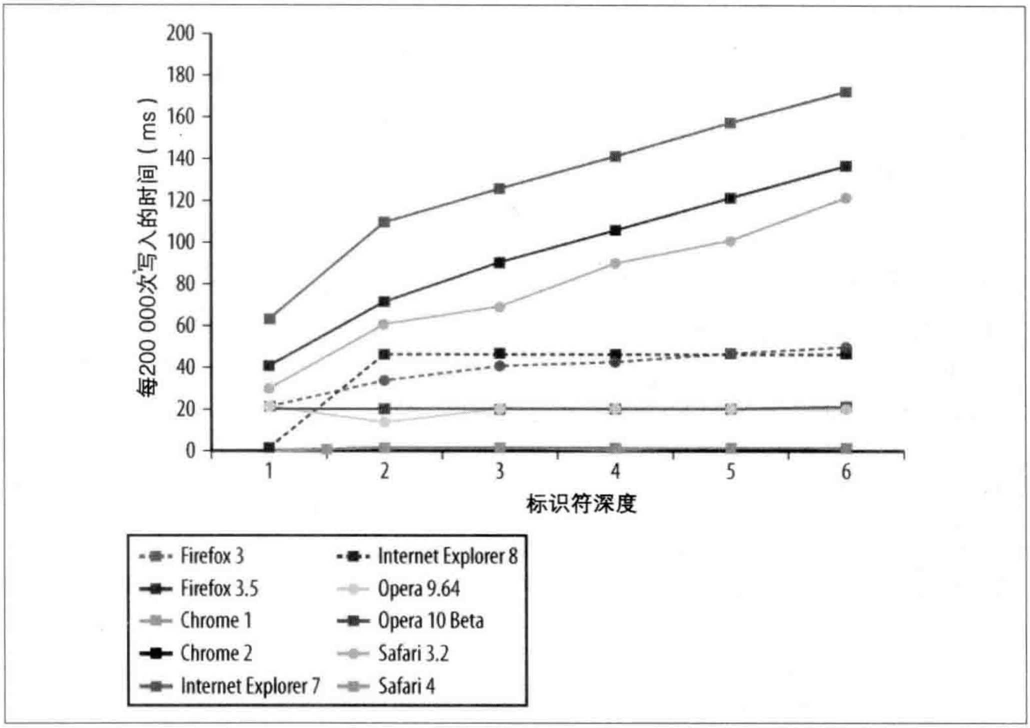


图 2-4　写操作的标识符解析速度

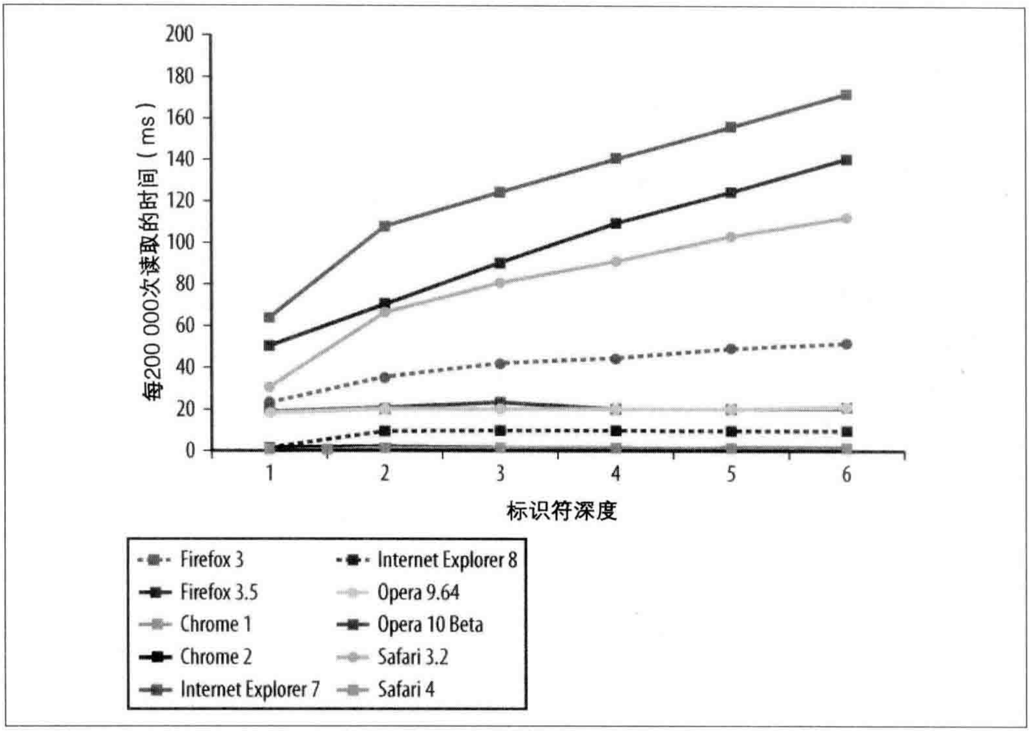


图 2-5　读操作的标识符解析速度

对所有浏览器而言，总的趋势是，一个标识符所在的位置越深，它的读写速度也就越慢。采用优化过的 JavaScript 引擎的浏览器，比如 Chrome 和 Safari 4，访问跨作用域的标识符时就没有类似的性能损失。然而其他浏览器，比如 IE，Safari 3.2 则受到严重影响。早期浏览器如 IE 6 和 Firefox 2，有令人难以置信的陡峭曲线，上图如果包括它们的数据，曲线最高点将超出图表边界。

综上所述，在没有优化 JavaScript 引擎的浏览器中，建议尽可能使用局部变量。一个好的经验法则是：如果某个跨作用域的值在函数中被引用一次以上，那么就把它存储到局部变量里。考虑下面的例子：

```
function initUI(){
    var bd = document.body,
        links = document.getElementsByTagName("a"),
        i= 0,
        len = links.length;

    while(i < len){
        update(links[i++]);
    }
    document.getElementById("go-btn").onclick = function(){
        start();
    };

    bd.className = "active";
}
```

该函数引用了三次 document，而 document 是个全局对象。搜索该变量的过程必须遍历整个作用域链，直到最后在全局变量对象中找到。你可以通过以下方法减少对性能的影响：先将全局变量的引用存储在一个局部变量中，然后使用这个局部变量代替全局变量。例如，上面的代码可以重写如下：

```
function initUI(){
    var doc = document,
        bd = doc.body,
        links = doc.getElementsByTagName("a"),
        i= 0,
        len = links.length;

    while(i < len){
        update(links[i++]);
    }

    doc.getElementById("go-btn").onclick = function(){
        start();
    };

    bd.className = "active";
}
```

更新后的 initUI()函数首先把 document 对象的引用存储到局部变量 doc 中，访问全局变量的次数由 3 次减少到 1 次。由于 doc 是个局部变量，因此通过它访问 document 会更快。当然，由于数量的原因，这个简单的函数不会显示出巨大的性能提升，但是可以想象，如果有几十个全局变量被反复访问，那么性能的改善将有多么显著。

改变作用域链

Scope Chain Augmentation

一般来说，一个执行环境的作用域链是不会改变的。但是，有两个语句可以在执行时临时改变作用域链。第一个是 with 语句。

With 语句用来给对象的所有属性创建了一个变量。在其他语言中，类似功能通常用来避免书写重复代码。函数 initUI()可以重写如下：

```
function initUI(){
    with (document){    // 避免!
        var bd = body,
            links = getElementsByTagName("a"),
            i= 0,
            len = links.length;

        while(i < len){
            update(links[i++]);
        }

        getElementById("go-btn").onclick = function(){
            start();
        };

        bd.className = "active";
    }
}
```

重写后的 `initUI()`版本使用 `with` 语句来避免多次书写 `document`。这看上去更高效，而实际上产生了一个性能问题。

当代码执行到 `with` 语句时，执行环境的作用域链临时被改变了。一个新的变量对象被创建，它包含了参数指定的对象的所有属性。这个对象被推入作用域链的首位，这意味着函数的所有局部变量现在处于第二个作用域链对象中，因此访问的代价更高了（参见图 2-6）。

通过把 `document` 对象传递给 `with` 语句，一个包含了 `document` 对象所有属性的新的可变变量被置于作用域链的头部。这使得访问 `document` 对象的属性非常快，而访问局部变量则变慢了，比如变量 `bd`。因此，最好避免使用 `with` 语句。综前所述，只要简单地把 `document` 存储在一个局部变量中，就可以提升性能。

在 JavaScript 中，并不是只有 `with` 语句能人为改变执行环境作用域链，`try-catch` 语句中的 `catch` 子句也具有同样的效果。当 `try` 代码块中发生错误，执行过程会自动跳转到 `catch` 子句，然后把异常对象推入一个变量对象并置于作用域的首位。在 `catch` 代码块内部，函数所有局部变量将会放在第二个作用域链对象中。示例如下：

```
try {
    methodThatMightCauseAnError();
} catch (ex){
    alert(ex.message);  // 作用域链在此处改变
}
```

请注意，一旦 `catch` 子句执行完毕，作用域链就会返回到之前的状态。

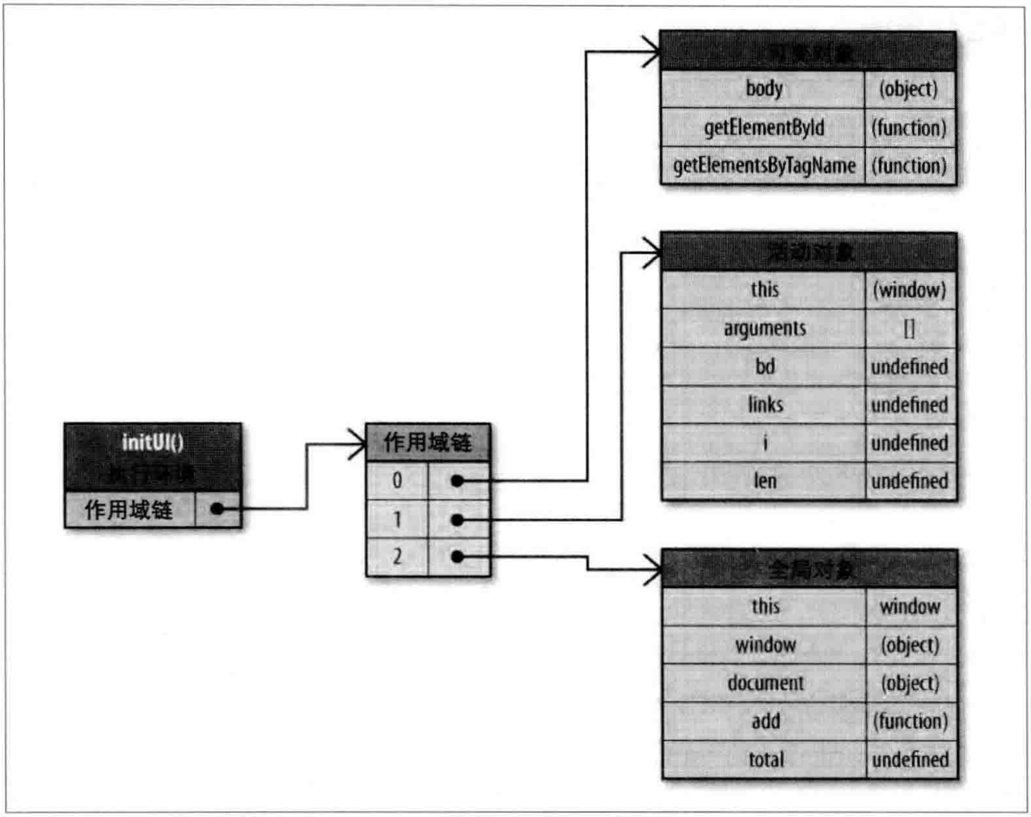

body
(object)
getElementById
(function)
getElementsByTagName
(function)
活动对象
this
(window)
arguments
[]
bd
undefined
links
undefined
i
undefined
len
undefined
initUI()
执行环境
作用域链
作用域链
0
1
2
全局对象
this
window
window
(object)
document
(object)
add
(function)
total
undefined

图 2-6　with 语句中改变后的作用域链

如果使用得当，`try-catch` 是个非常有用的语句，因此不建议完全弃用。如果你准备使用 `try-catch`，请确保了解可能会出现的错误。`try-catch` 语句不应该被用来解决 JavaScript 错误。如果你知道某个错误经常出现，那说明是代码本身有问题，应该尽早被修复。

你尽量简化代码来使得 `catch` 子句对性能的影响最小化。一种推荐的做法是将错误委托给一个函数来处理，比如下面的例子：

```
try {
    methodThatMightCauseAnError();
} catch (ex){
    handleError(ex); // 委托给错误处理器函数
}
```

函数 `handleError()`是 `catch` 子句中唯一执行的代码。该函数接受错误产生的异常对象为参数，你可以用适当的方式灵活地处理错误。由于只执行一条语句，且没有局部变量的访问，作用域链的临时改变就不会影响代码性能。

动态作用域

Dynamic Scopes

无论是 `with` 语句还是 `try-catch` 语句的 `catch` 子句，或是包含 `eval()`的函数，都被认为是动态作用域。动态作用域只存在于代码执行过程中，因此无法通过静态分析（查看代码结构）检测出来。例如：

```
function execute(code) {
  eval(code);

  function subroutine(){
    return window;
  }

  var w = subroutine();

  // w 是什么?
};
```

由于使用了 `eval()`，函数 `execute()`看上去像动态作用域。变量 `w` 的值会随 `code` 的值改变。大部分情况下，`w` 等同于全局的 `window` 对象，但是考虑如下情况：

```
execute("var window = {};")
```

以上代码中，`execute()`中的 `eval()`创建了一个局部变量 `window`，因此 `w` 等同于局部变量 `window` 而非全局的 `window` 对象。只有执行这段代码时才会发现问题，这意味着 `window` 标识符的真实值是无法被预知的。

经过优化的 JavaScript 引擎，比如 Safari 的 Nitro 引擎，尝试通过分析代码来确定哪些变量可以在特定时候被访问。这些引擎试图避开传统作用域链的查找，取代以标识符索引的方式进行快速查找。当涉及动态作用域时，这种优化方式就失效了。脚本引擎必须切换回较慢的基于哈希表的标识符识别方式，这更像是传统的作用域链查找。

因此，只有在确实有必要时才推荐使用动态作用域。

闭包、作用域和内存

Closures, Scope, and Memory

闭包是 JavaScript 最强大的特性之一，它允许函数访问局部作用域之外的数据。闭包的使用通过 Douglas Crockford 的多篇文章的介绍而流行开来，如今普遍应用在复杂的 Web 应用中。然而，使用闭包可能会导致性能问题。

思考以下代码，尝试理解与闭包有关的性能问题：

```
function assignEvents(){

    var id = "xdi9592";

    document.getElementById("save-btn").onclick = function(event){
        saveDocument(id);
    };
}
```

`assignEvents()`函数给一个 DOM 元素设置事件处理函数。这个事件处理函数就是一个闭包，它在 `assignEvenets()`执行时创建，并且能访问所属作用域的 `id` 变量。为了让这个闭包访问 `id`，必须创建一个特定的作用域链。

当 `assignEvents()`函数执行时，一个包含了变量 `id` 以及其他数据的活动对象被创建。它成为执行环境作用域链中的第一个对象，而全局对象紧随其后。当闭包被创建时，它的`[[Scope]]`属性被初始化为这些对象（参见图 2-7）。

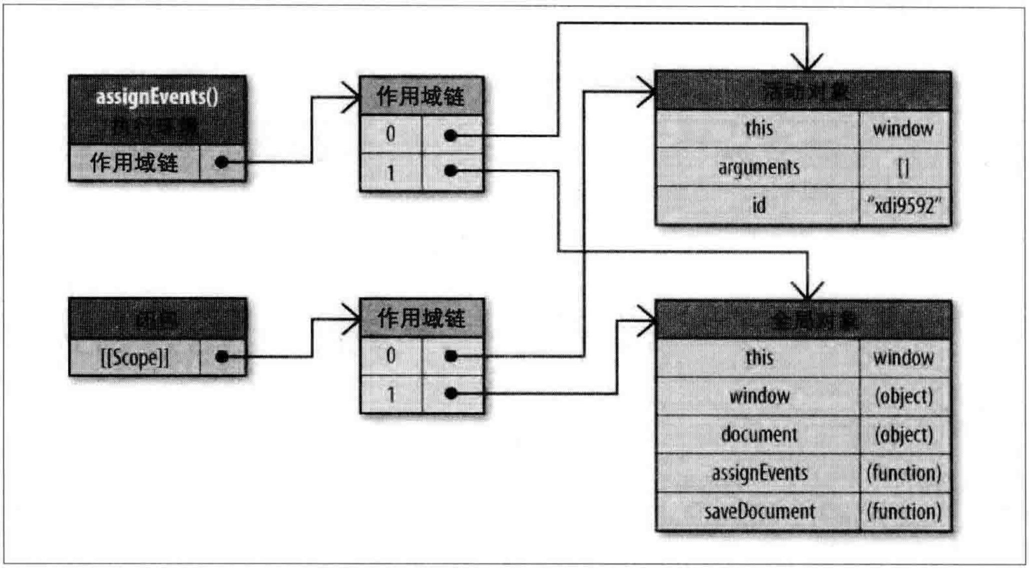

assignEvents()
执行环境
作用域链
作用域链
0
1
活动对象
this
window
arguments
[]
id
"xdi9592"
闭包
[[Scope]]
作用域链
0
1
全局对象
this
window
window
(object)
document
(object)
assignEvents
(function)
saveDocument
(function)

图 2-7　函数 assignEvents()执行环境的作用域链和闭包

由于闭包的`[[Scope]]`属性包含了与执行环境作用域链相同的对象的引用，因此会产生副作用。通常来说，函数的活动对象会随着执行环境一同销毁。但引入闭包时，由于引用仍然

存在于闭包的`[[Scope]]`属性中，因此激活对象无法被销毁。这意味着脚本中的闭包与非闭包函数相比，需要更多的内存开销。在大型 Web 应用中，这可能是个问题，尤其在 IE 浏览器中需要关注。由于 IE 使用非原生 JavaScript 对象来实现 DOM 对象，因此闭包会导致内存泄漏（详见第 3 章）。

当闭包代码执行时，会创建一个执行环境，它的作用域链与属性`[[Scope]]`中所引用的两个相同的作用域链对象一起被初始化，然后一个活动对象为闭包自身所创建（参见图 2-8）。

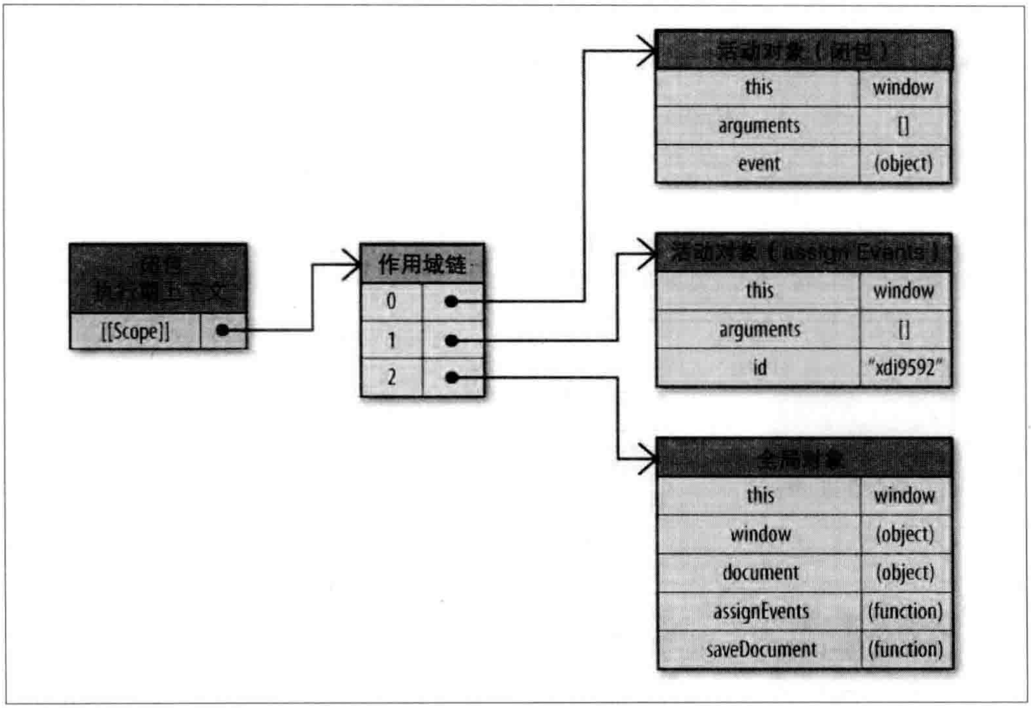

活动对象（闭包）
this
window
arguments
[]
event
(object)
闭包
执行期上下文
[[Scope]]
作用域链
0
1
2
活动对象（assignEvents）
this
window
arguments
[]
id
"xdi9592"
全局对象
this
window
window
(object)
document
(object)
assignEvents
(function)
saveDocument
(function)

图 2-8　运行闭包

注意在闭包中用到的两个标识符，`id` 和 `saveDocument`，它们的位置在作用域链第一个对象之后。这就是使用闭包最需要关注的性能点：在频繁访问跨作用域的标识符时，每次访问都会带来性能损失。

在脚本编程中，最好小心地使用闭包，它同时关系到内存和执行速度。不过，你可以遵照本章节中早先对跨作用域变量的处理建议，来减轻闭包对执行速度的影响：将常用的跨作用域变量存储在局部变量中，然后直接访问局部变量。

对象成员
Object Members

大部分 JavaScript 代码是以面向对象风格编写的，无论是通过创建自定义对象还是使用内置对象(比如文档对象模型（DOM）和浏览器对象模型（BOM）中的对象)。这会导致非常频繁地访问对象成员。

对象成员包括属性和方法，在 JavaScript 中，二者有些许差异。一个被命名的对象成员能包含任何数据类型。既然函数也是一种对象，那么对象成员除传统数据类型外，还可以包含函数。当一个被命名的成员引用了一个函数，该成员就被称为一个“方法”，相反，引用了非函数类型的成员就被称为“属性”。

本章前部分讨论到，访问对象成员的速度比访问字面量或变量要慢，在某些浏览器中比访问数组元素还要慢。为了理解其中的原因，有必要先理解 JavaScript 中对象的本质。

原型
Prototypes

JavaScript 中的对象是基于原型的。原型是其他对象的基础，它定义并实现了一个新创建的对象所必须包含的成员列表。这一概念完全不同于传统面向对象编程语言中的“类”的概念，“类”定义了创建新对象的过程，而原型对象为所有对象实例所共享，因此这些实例也共享了原型对象的成员。

对象通过一个内部属性绑定到它的原型。在 Firefox，Safari 和 Chrome 浏览器中，这个属性 `__proto__`对开发者可见，而其他的浏览器却不允许脚本访问此属性。一旦你创建一个内置对象（比如 `Object` 和 `Array`）的实例，它们就会自动拥有一个 `Object` 实例作为原型。

因此，对象可以有两种成员类型：实例成员（也称为 own 成员）和原型成员。实例成员直接存在于对象实例中，原型成员则从对象原型继承而来。考虑以下示例：

```
var book = {
    title: "High Performance JavaScript",
    publisher: "Yahoo! Press"
};

alert(book.toString());  //"[object Object]"
```

在这段代码中，对象 book 有两个实例成员：title 和 publisher。注意其中并没有定义 toString() 方法，但这个方法却被顺利执行，也没有抛出错误。方法 toString()是由对象 book 继承而来的原型成员。图 2-9 展示了它们之间的关系。

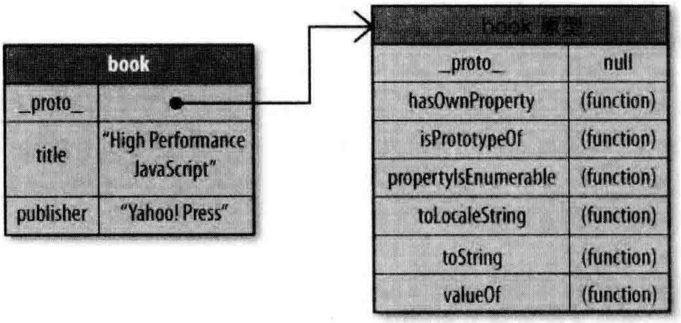

book
proto
title
"High Performance JavaScript"
publisher
"Yahoo! Press"
book 原型
proto
null
hasOwnProperty
(function)
isPrototypeOf
(function)
propertyIsEnumerable
(function)
toLocaleString
(function)
toString
(function)
valueOf
(function)

图 2-9　实例和原型的关系

解析对象成员的过程与解析变量十分相似。当 book.toString()被调用时，会从对象实例开始，搜索名为“toString”的成员。一旦 book 没有名为 toString 的成员，那么会继续搜索其原型对象，直到 toString()方法被找到并且执行。由此可见，对象 book 可以访问它原型中的每一个属性和方法。

你可以用 hasOwnProperty()方法来判断对象是否包含特定的实例成员（传递给方法的参数名即成员的名称）。要确定对象是否包含特定的属性，可以使用 in 操作符。例如：

```
var book = {
    title: "High Performance JavaScript",
    publisher: "Yahoo! Press"
};

alert(book.hasOwnProperty("title"));    //true
alert(book.hasOwnProperty("toString")); //false

alert("title" in book);       //true
alert("toString" in book);    //true
```

在这段代码中，由于 title 是个实例成员，因此将参数“title”传给 hasOwnProperty()时会返回 true，而传入参数“toString”时返回 false，因为该实例中并不存在此成员。使用 in 操作符时，两种情况下都会返回 true，因为它会既搜索实例也搜索原型。

原型链

Prototype Chains

对象的原型决定了实例的类型。默认情况下，所有对象都是对象（Object）的实例，并继承了所有基本方法，比如 toString()。你可以定义并使用构造函数来创建另一种类型的原型。例如：

```
function Book(title, publisher){
    this.title = title;
    this.publisher = publisher;
}

Book.prototype.sayTitle = function(){
    alert(this.title);
};

var book1 = new Book("High Performance JavaScript", "Yahoo! Press");
var book2 = new Book("JavaScript: The Good Parts", "Yahoo! Press");

alert(book1 instanceof Book);      //true
alert(book1 instanceof Object);    //true

book1.sayTitle();         //"High Performance JavaScript"
alert(book1.toString()); //"[object Object]"
```

使用构造函数 Book 来创建一个新的 Book 实例。实例 book1 的原型（__proto__）是 Book.prototype，而 Book.prototype 的原型是 Object。这是原型链的创建过程，book1 和 book2 继承了原型链中的所有成员。图 2-10 显示了这个关系。

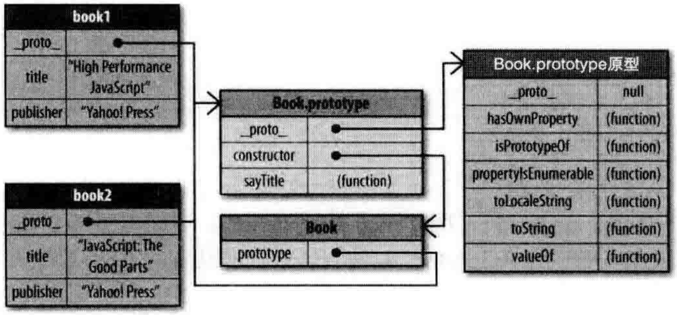

book1
proto
title
"High Performance JavaScript"
publisher
"Yahoo! Press"
book2
proto
title
"JavaScript: The Good Parts"
publisher
"Yahoo! Press"
Book.prototype
proto
constructor
sayTitle
(function)
Book
prototype
Book.prototype原型
proto
null
hasOwnProperty
(function)
isPrototypeOf
(function)
propertyIsEnumerable
(function)
toLocaleString
(function)
toString
(function)
valueOf
(function)

图 2-10　原型链

注意，这两个 Book 实例共享着同一个原型链，它们有着各自的 title 和 publisher 属性，而其他部分都继承自原型。

当调用 `book1.toString()`时，搜索过程会深入原型链中直到找到对象成员“toString”。你也许会猜到，对象在原型链中存在的位置越深，找到它也就越慢。图 2-11 显示了对象成员的深度和访问它所需时间的关系。

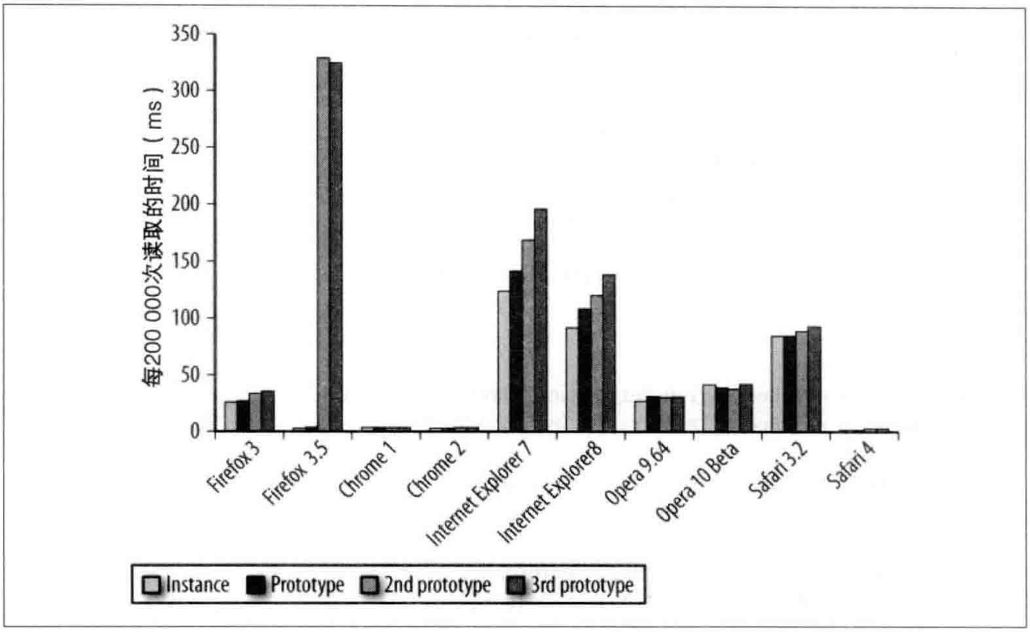


图 2-11　原型链中的数据访问深度

尽管使用优化过的 JavaScript 引擎的新型浏览器在此过程表现优异，但是老版本的浏览器，特别是 IE 和 Firefox 3.5，每深入一层原型链都会增加性能损失。请记住，搜索实例成员比从字面量或局部变量中读取数据代价更高，再加上遍历原型链带来的开销，这让性能问题更为严重。

嵌套成员

Nested Members

由于对象成员可能包含其他成员，例如不太常见的写法：`window.location.href`。每次遇到点操作符，嵌套成员会导致 JavaScript 引擎搜索所有对象成员。图 2-12 显示了对象成员深度和读取时间的关系。

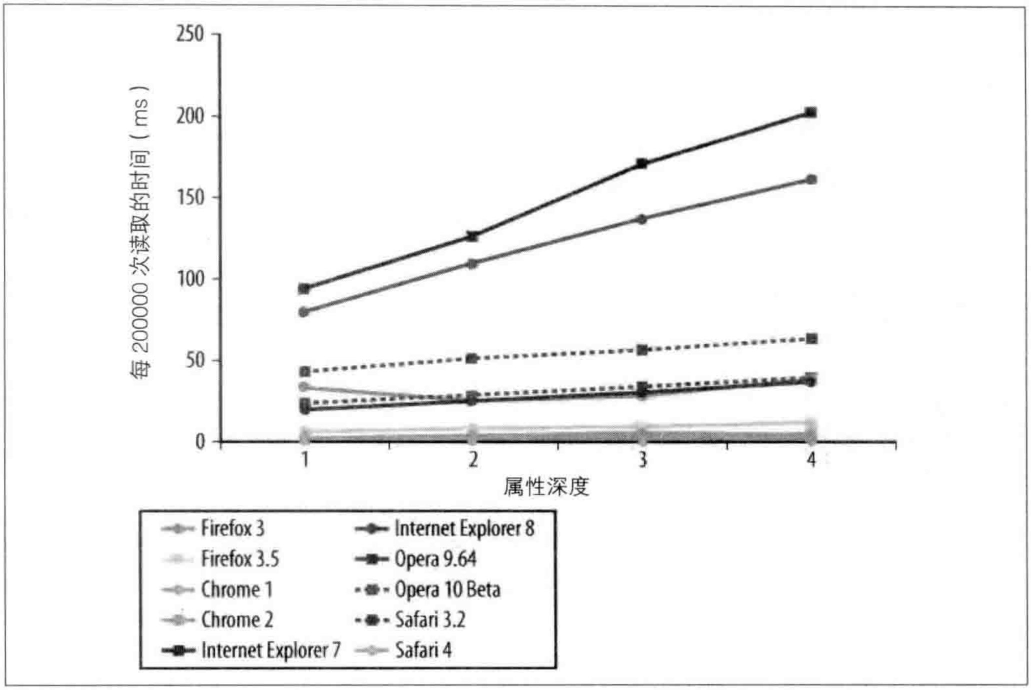


图 2-12　读取时间和属性深度的关系

结果不出所料，对象成员嵌套得越深，读取速度就会越慢。执行 `location.href` 总是比 `window.location.href` 要快，后者也比 `window.location.href.toString()`要快。如果这些属性不是对象的实例属性，那么成员解析还需要搜索原型链，这会花更多的时间。

提示：大部分浏览器中，通过点表示法（`object.name`）操作和通过括号表示法（`object["name"]`）操作并没有明显的区别。只有在 Safari 中，点符号始终会更快，但这并不意味着不要用括号符号。

缓存对象成员值

Caching Object Member Values

由于所有类似的性能问题都与对象成员有关，因此应该尽可能避免使用它们。更确切地说，应当注意，只在必要时使用对象成员。例如，在同一个函数中没有必要多次读取同一个对象成员。

```
function hasEitherClass(element, className1, className2){
    return element.className == className1 || element.className == className2;
}
```

以上代码中，`element.className` 被读取了两次。很显然，在该函数语句中它的值并未改变，却仍然执行了两次对象成员查找。你可以将值保存在局部变量中来减少一次查找，如以下代码：

```
function hasEitherClass(element, className1, className2){
    var currentClassName = element.className;
    return currentClassName == className1 || currentClassName == className2;
}
```

重写后的版本中将对象成员的查找次数减少到 1 次。既然两次成员查找都是通过读取属性值，那么有必要在第一次查找到值后就将其存储在局部变量中，因为局部变量的读取速度要快得多。

通常来说，在函数中如果要多次读取同一个对象属性，最佳做法是将属性值保存到局部变量中。局部变量能用来替代属性以避免多次查找带来的性能开销。特别是在处理嵌套对象成员时，这样做会明显提升执行速度。

JavaScript 的命名空间，比如 YUI 中使用的技术，是导致频繁访问嵌套属性的起因之一。例如：

```
function toggle(element){
    if (YAHOO.util.Dom.hasClass(element, "selected")){
        YAHOO.util.Dom.removeClass(element, "selected");
        return false;
    } else {
        YAHOO.util.Dom.addClass(element, "selected");
        return true;
    }
}
```

这段代码中重复读取 `YAHOO.util.Dom` 3 次来读取 3 个不同的方法。每个方法又有 3 次成员查找，一共就有 9 次，导致代码十分低效。更好的做法是将 `YAHOO.util.Dom` 保存在局部变量中，再访问该局部变量：

```
function toggle(element){
    var Dom = YAHOO.util.Dom;
    if (Dom.hasClass(element, "selected")){
        Dom.removeClass(element, "selected");
        return false;
    } else {
        Dom.addClass(element, "selected");
        return true;
    }
}
```

代码中对象成员读取次数由 9 次减少到 5 次。请不要在同一个函数里多次查找同一个对象成员，除非它的值改变了。

提示：这种优化技术并不推荐用于对象的成员方法。因为许多对象方法使用 this 来判断执行环境，把一个对象方法保存在局部变量会导致 this 绑定到 window，而 this 值的改变会使得 JavaScript 引擎无法正确解析它的对象成员，进而导致程序出错。

小结

Summary

在 JavaScript 中，数据存储的位置会对代码整体性能产生重大的影响。数据存储共有 4 种方式：字面量、变量、数组项、对象成员。它们有着各自的性能特点。

- 访问字面量和局部变量的速度最快，相反，访问数组元素和对象成员相对较慢。
- 由于局部变量存在于作用域链的起始位置，因此访问局部变量比访问跨作用域变量更快。变量在作用域链中的位置越深，访问所需时间就越长。由于全局变量总处在作用域链的最末端，因此访问速度也是最慢的。
- 避免使用 `with` 语句，因为它会改变执行环境作用域链。同样，`try-catch` 语句中的 `catch` 子句也有同样的影响，因此也要小心使用。
- 嵌套的对象成员会明显影响性能，尽量少用。
- 属性或方法在原型链中的位置越深，访问它的速度也越慢。
- 通常来说，你可以通过把常用的对象成员、数组元素、跨域变量保存在局部变量中来改善 JavaScript 性能，因为局部变量访问速度更快。

通过以上策略，你可以显著提升那些需要使用大量 JavaScript 的 Web 应用的实际性能。

第 3 章

DOM 编程

DOM Scripting

Stoyan Stefanov

用脚本进行 DOM 操作的代价很昂贵，它是富 Web 应用中最常见的性能瓶颈。本章将讨论可能对程序造成负面影响的 DOM 编程，并给出提高响应速度的建议。本章讨论以下三类问题：

- 访问和修改 DOM 元素。
- 修改 DOM 元素的样式会导致重绘（repaint）和重排（reflow）。
- 通过 DOM 事件处理与用户的交互。

首先——什么是 DOM？它为什么慢？

浏览器中的 DOM

DOM in the Browser World

文档对象模型（DOM）是一个独立于语言的，用于操作 XML 和 HTML 文档的程序接口（API）。在浏览器中，主要用来与 HTML 文档打交道，同样也用在 Web 程序中获取 XML 文档，并使用 DOM API 用来访问文档中的数据。

尽管 DOM 是个与语言无关的 API，它在浏览器中的接口却是用 JavaScript 实现的。客户端脚本编程大多数时候是在和底层文档（underlying document）打交道，DOM 就成为现在 JavaScript 编码中的重要部分。

浏览器中通常会把 DOM 和 JavaScript 独立实现。比如在 IE 中，JavaScript 的实现名为 JScript，位于 jscript.dll 文件中；DOM 的实现则存在另一个库中，名为 mshtml.dll（内部称为 Trident）[译注1]。这个分离允许其他技术和语言，比如 VBScript，能共享使用 DOM 以及 Trident

---

译注1：参见维基百科 *http://zh.wikipedia.org/zh/Trident*（%E6%BE%92%E7%89%88%E5%BC%95%E6%93% 8E）。

提供的渲染函数。Safari 中的 DOM 和渲染是使用 Webkit 中的 WebCore 实现，JavaScript 部分是由独立的 JavaScriptCore 引擎（最新版本的名字为 SquirrelFish）来实现。Google Chrome 同样使用 WebKit 中的 WebCore 库来渲染页面，但 JavaScript 引擎是他们自己研发的，名为 V8。Firefox 的 JavaScript 引擎名为 SpiderMonkey（最新版的名字为 TraceMonkey）[译注2]，与名为 Gecko 的渲染引擎相互独立。

天生就慢

Inherently Slow

这对性能意味着什么？简单理解，两个相互独立的功能只要通过接口彼此连接，就会产生消耗。有个贴切的比喻，把 DOM 和 JavaScript（这里指 ECMAScript）各自想象为一个岛屿，它们之间用收费桥梁连接（参考微软 MIX09 会议 John Hrvatin 的演讲，*http://videos.visitmix.com/MIX09/T53F*）。ECMAScript 每次访问 DOM，都要途经这座桥，并交纳“过桥费”。访问 DOM 的次数越多，费用也就越高。因此，推荐的做法是尽可能减少过桥的次数，努力待在 ECMAScript 岛上。本章剩余部分将着重解释这个过程的具体原因，以及寻找相应方法来提升页面中的交互响应速度。

DOM 访问与修改

DOM Access and Modification

访问 DOM 元素是有代价的——前面提到的“过桥费”。修改元素则更为昂贵，因为它会导致浏览器重新计算页面的几何变化。

当然，最坏的情况是在循环中访问或修改元素，尤其是对 HTML 元素集合循环操作。

为了让你对 DOM 编程带来的性能问题有个量化的了解，请看下面的简单实例：

```
function innerHTMLLoop() {
    for (var count = 0; count < 15000; count++) {
        document.getElementById('here').innerHTML += 'a';
    }
}
```

这个函数循环修改页面元素的内容。这段代码的问题在于，每次循环迭代，该元素都被访问两次：一次读取 `innerHTML` 属性值，另一次重写它。

换一种效率更高的方法，用局部变量存储修改中的内容，在循环结束后一次性写入：

译注2：　更新的版本为 JagerMonkey。

```
function innerHTMLLoop2() {
    var content = '';
    for (var count = 0; count < 15000; count++) {
        content += 'a';
    }
    document.getElementById('here').innerHTML += content;
}
```

在所有浏览器中，修改后的版本都运行得更快。图 3-1 显示在不同浏览器中检测到的速度提升。图中 *y* 轴（如同本章所有图片）显示了运行速度的提升情况，比如，一个比另一个快了多少倍。例如在 IE 6 中，使用 `innerHTMLLoop2()`比用 `innerHTMLLoop()`快 155 倍。

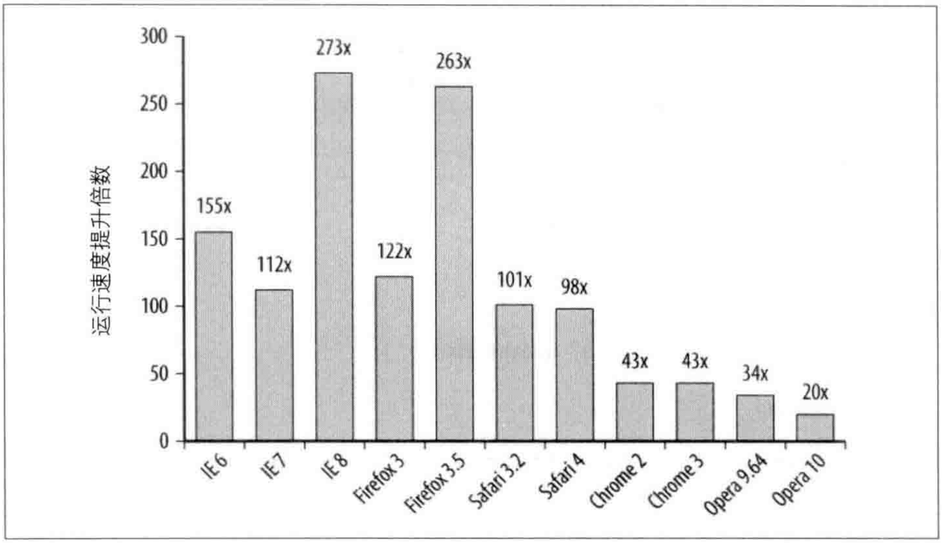


图 3-1　把运算留给 ECMAScript 这一端处理带来的一个好处是：innerHTMLLoop2()比 innerHTMLLoop()快上百倍

结果显而易见，访问 DOM 的次数越多，代码的运行速度越慢。因此，通用的经验法则是：减少访问 DOM 的次数，把运算尽量留在 ECMAScript 这一端处理。

innerHTML 对比 DOM 方法

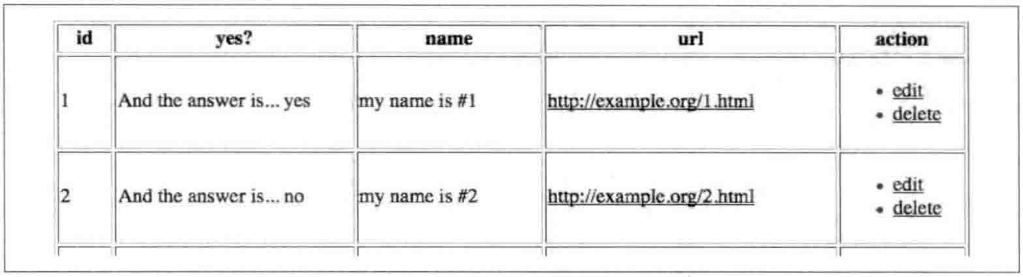
innerHTML Versus DOM methods

多年来，在 Web 开发社区中围绕着这个问题有着许多讨论：修改页面区域的最佳方案是用非标准但支持良好的 `innerHTML` 属性呢？还是只用类似 `document.createElement()`的原生 DOM 方法？若不考虑 Web 标准，那么性能如何？答案是：相差无几。但是，在除开最新版的 WebKit 内核（Chrome 和 Safari）之外的所有浏览器中，`innerHTML` 会更快一些。

让我们来看个例子，用两种方法创建一个 1000 行的表格：

- 合并 HTML 字符，然后更新 DOM 的 `innerHTML` 属性。
- 只用标准的 DOM 方法，比如 `document.createElement()`和 `document.createTextNode()`。

例子中的表格内容很像来自某一个内容管理系统（Content Management System），结果显示在图 3-2 中。



id	yes?	name	url	action
1	And the answer is... yes	my name is #1	http://example.org/1.html	edit delete
2	And the answer is... no	my name is #2	http://example.org/2.html	edit delete

图 3-2 最终生成一个 5 列 1000 行的 HTML 表格

使用 `innerHTML` 生成表格的代码如下：

```
function tableInnerHTML() {
    var i, h = ['<table border="1" width="100%">'];

    h.push('<thead>');

h.push('<tr><th>id<\/th><th>yes?<\/th><th>name<\/th><th>url<\/th><th>action<\/th>
<\/tr>');
    h.push('<\/thead>');
    h.push('<tbody>');
    for (i = 1; i <= 1000; i++) {
        h.push('<tr><td>');
        h.push(i);
        h.push('<\/td><td>');
        h.push('And the answer is... ' + (i % 2 ? 'yes' : 'no'));
        h.push('<\/td><td>');
        h.push('my name is #' + i);
        h.push('<\/td><td>');
        h.push('<a href="http://example.org/' + i + '.html">http:// example. org/'
+ i + '.html<\/a>');
        h.push('<\/td><td>');
        h.push('<ul>');
        h.push(' <li><a href="edit.php?id=' + i + '">edit<\/a><\/li>');
        h.push(' <li><a href="delete.php?id="' + i + '-id001">delete <\/a><\/li>');
        h.push('<\/ul>');
        h.push('<\/td>');
        h.push('<\/tr>');
        }

        h.push('<\/tbody>');
        h.push('<\/table>');

        document.getElementById('here').innerHTML = h.join('');
    };
```

使用 DOM 方法生成相同的表格，代码显得更冗长：

```
    function tableDOM() {

        var i, table, thead, tbody, tr, th, td, a, ul, li;

        tbody = document.createElement('tbody');

        for (i = 1; i <= 1000; i++) {

            tr = document.createElement('tr');
            td = document.createElement('td');
            td.appendChild(document.createTextNode((i % 2) ? 'yes' : 'no'));
            tr.appendChild(td);
            td = document.createElement('td');
            td.appendChild(document.createTextNode(i));
            tr.appendChild(td);
            td = document.createElement('td');
            td.appendChild(document.createTextNode('my name is #' + i));
            tr.appendChild(td);

            a = document.createElement('a');
            a.setAttribute('href', 'http://example.org/' + i + '.html');
            a.appendChild(document.createTextNode('http://example.org/' + i +'.html'));
            td = document.createElement('td');
            td.appendChild(a);
            tr.appendChild(td);

            ul = document.createElement('ul');
            a = document.createElement('a');
            a.setAttribute('href', 'edit.php?id=' + i);
            a.appendChild(document.createTextNode('edit'));
            li = document.createElement('li');
            li.appendChild(a);
            ul.appendChild(li);
            a = document.createElement('a');
            a.setAttribute('href', 'delete.php?id=' + i);
            a.appendChild(document.createTextNode('delete'));
            li = document.createElement('li');
            li.appendChild(a);
            ul.appendChild(li);
            td = document.createElement('td');
            td.appendChild(ul);
            tr.appendChild(td);

            tbody.appendChild(tr);
    }

    tr = document.createElement('tr');
    th = document.createElement('th');
    th.appendChild(document.createTextNode('yes?'));
    tr.appendChild(th);
    th = document.createElement('th');
    th.appendChild(document.createTextNode('id'));
    tr.appendChild(th);
    th = document.createElement('th');
    th.appendChild(document.createTextNode('name'));
    tr.appendChild(th);
    th = document.createElement('th');
    th.appendChild(document.createTextNode('url'));
    tr.appendChild(th);
    th = document.createElement('th');
    th.appendChild(document.createTextNode('action'));
    tr.appendChild(th);

    thead = document.createElement('thead');
    thead.appendChild(tr);
    table = document.createElement('table');
    table.setAttribute('border', 1);
    table.setAttribute('width', '100%');
    table.appendChild(thead);
    table.appendChild(tbody);

    document.getElementById('here').appendChild(table);
};
```

图 3-3 中显示了分别使用 `innerHTML` 和原生 DOM 方法生成 HTML 表格的结果对比。在旧版本的浏览器中，`innerHTML` 的优势更加明显（IE 6 中 `innerHTML` 的效率比原生 DOM 方法快 3.6 倍），但在新版本中优势则不那么明显。在基于 WebKit 内核的新版浏览器中恰恰相反：使用 DOM 方法略胜一筹。因此，最终选择哪种方式取决于你的用户经常使用的浏览器，以及你的编码习惯。

提示：补充说明，请记住本例中使用的字符串合并在老版本 IE 下性能并不是最好的（参考第 5 章）。建议使用数组来合并大量字符串，这样会让 `innerHTML` 效率更高。

如果在一个对性能有着苛刻要求的操作中更新一大段 HTML，推荐使用 `innerHTML`，因为它在绝大部分浏览器中都运行得更快。但对大多数日常操作而言，并没有太大区别，所以你更应该根据可读性、稳定性、团队习惯、代码风格来综合决定使用哪种方式。

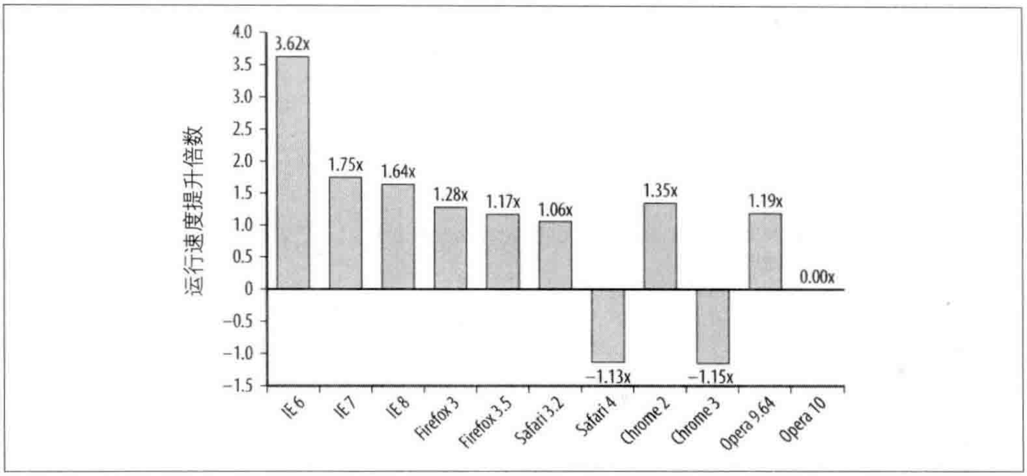


图 3-3　对比使用 innerHTML 和 DOM 方法来创建 1000 行的表格；在 IE 6 中，innerHTML 要快 3 倍以上，而在最新版 WebKit 浏览器中 innerHTMC 则会相对较慢。

节点克隆

Cloning Nodes

使用 DOM 方法更新页面内容的另一个途径是克隆已有元素，而不是创建新元素——换句话说，就是使用 `element.cloneNode()`（`element` 表示已有节点）替代 `document. createElement()`。

在大多数浏览器中，节点克隆都更有效率，但也不是特别明显。用下面的方法重新构建前面示例中的表格，先创建需要重复的元素，然后重复拷贝操作，这样做的运行结果只稍快一点：

- 在 IE 8 中快 2%，IE 6 和 IE 7 没变化。
- 在 Firefox 3.5 和 Safari 4 中快了 5.5%。
- 在 Opera 中快了 6%（但在 Opera 10 中无变化）。
- Chrome 2 快了 10%，Chrome 3 为 3%。[译注3]

作为示例，下面是用 `element.cloneNode()` 生成表格的部分代码：

```
function tableClonedDOM() {

    var i, table, thead, tbody, tr, th, td, a, ul, li,
        oth = document.createElement('th'),
        otd = document.createElement('td'),
        otr = document.createElement('tr'),
        oa  = document.createElement('a'),
        oli = document.createElement('li'),
```

[译注3] 在最新版的 Chrome 43 和 Safari 8 下测试，结果为 tableClonedDOM 函数执行速度是 tableDOM 的 3~5 倍，是 tableInnerHTML 的 5~10 倍。

```
        oul = document.createElement('ul');

    tbody = document.createElement('tbody');

    for (i = 1; i <= 1000; i++) {

        tr = otr.cloneNode(false);
        td = otd.cloneNode(false);
        td.appendChild(document.createTextNode((i % 2) ? 'yes' : 'no'));
        tr.appendChild(td);
        td = otd.cloneNode(false);
        td.appendChild(document.createTextNode(i));
        tr.appendChild(td);
        td = otd.cloneNode(false);
        td.appendChild(document.createTextNode('my name is #' + i));
        tr.appendChild(td);

        // 循环的剩余部分……

    }

    // 生成表格的其他部分……
}
```

HTML 集合

HTML Collections

HTML 集合是包含了 DOM 节点引用的类数组对象。以下方法的返回值就是一个集合：

- `document.getElementsByName()`
- `document.getElementsByClassName()`
- `document.getElementsByTagName()`

下面的属性同样返回 HTML 集合：

`document.images`
　　页面中所有 `img` 元素

`document.links`
　　所有 `a` 元素

`document.forms`
　　所有表单元素

`document.forms[0].elements`
　　页面中第一个表单的所有字段

以上方法和属性的返回值为 HTML 集合对象，这是个类似数组的列表。它们并不是真正的数组（因为没有 `push()`或 `slice()`之类的方法），但提供了一个类似数组中的 `length` 属性，并且还能以数字索引的方式访问列表中的元素。例如，`document.images[1]`返回集合中第二

个元素。正如 DOM 标准中所定义的，HTML 集以一种“假定实时态”（assumed to be live）实时存在，这意味着当底层文档对象更新时，它也会自动更新（参考 *http://www.w3.org/TR/DOM-Level- 2-HTML/html.html#ID-75708506*）。

事实上，HTML 集合一直与文档保持着连接，每次你需要最新的信息时，都会重复执行查询的过程，哪怕只是获取集合里的元素个数（即访问集合的 `length` 属性）也是如此。这正是低效之源。

昂贵的集合

为了演示集合的实时性，考虑以下代码片段：

```
// 一个意外的死循环
var alldivs = document.getElementsByTagName('div');
for (var i = 0; i < alldivs.length; i++) {
    document.body.appendChild(document.createElement('div'))
}
```

这段代码看上去只是简单地把页面中 div 元素数量翻倍。它遍历现有的 `div` 元素，每次创建一个新的 `div` 并添加到 `body` 中。但事实上这是一个死循环，因为循环的退出条件 `alldivs.length` 在每次迭代时都会增加，它反映出的是底层文档的当前状态。

像这样遍历 HTML 集合可能会导致逻辑错误，而且也很慢，因为每次迭代都执行查询操作（参见图 3-4）。

正如我们在第 4 章中将会讨论的，在循环的条件控制语句中读取数组的 `length` 属性是不推荐的做法。读取一个集合的 `length` 比读取普通数组的 `length` 要慢很多，因为每次都要重新查询。下面的示例说明了这个问题，它接收一个集合 `coll`，把它复制到数组 `arr` 中，然后对比多次执行所消耗的时间。

考虑这个函数，它将一个 HTML 集合拷贝到普通数组：

```
function toArray(coll) {
    for (var i = 0, a = [], len = coll.length; i < len; i++) {
        a[i] = coll[i];
    }
    return a;
}
```

设置一个集合，并把它拷贝到一个数组中：

```
var coll = document.getElementsByTagName('div');
var ar   = toArray(coll);
```

比较下面两个函数：

```
// 较慢
function loopCollection() {
    for (var count = 0; count < coll.length; count++) {
        /* 代码处理 */
    }
}

// 较快
function loopCopiedArray() {
    for (var count = 0; count < arr.length; count++) {
        /* 代码处理 */
    }
}
```

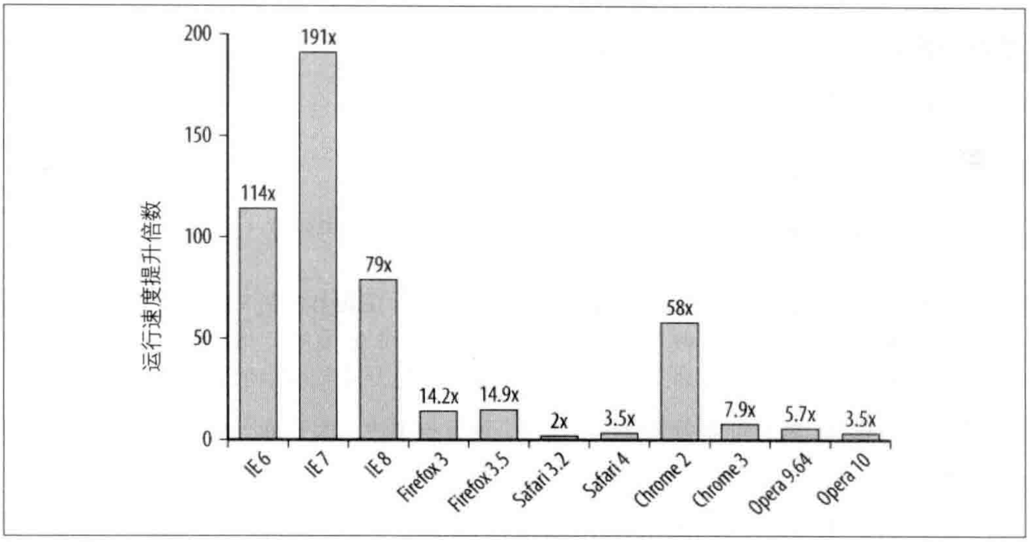


图 3-4　在相同的内容和数量下，遍历一个数组的速度明显快于遍历一个 HTML 集合

在每次迭代过程中，读取元素集合的 `length` 属性会引发集合进行更新，这在所有浏览器中都有明显的性能问题。优化方法很简单，把集合的长度缓存到一个局部变量中，然后在循环的条件退出语句中使用该变量：

```
function loopCacheLengthCollection() {
    var coll = document.getElementsByTagName('div'),
        len = coll.length;
    for (var count = 0; count < len; count++) {
        /* 代码处理 */
    }
}
```

此函数运行速度和 `loopCopiedArray()`一样快。

很多情况下如果只需要遍历一个相对较小的集合，那么缓存 `length` 就够了。但由于遍历数组比遍历集合快，因此如果先将集合元素拷贝到数组中，那么访问它的属性会更快。请记住，这会因额外的步骤带来消耗，而且会多遍历一次集合，因此应当评估在特定条件下，使用数组拷贝是否有帮助。

前面提到的 `toArray()`函数可作为一个通用的集合转数组函数。

访问集合元素时使用局部变量

前面的例子只是使用了一个空循环，如果要在循环中访问集合元素，会发生什么？

一般来说，对于任何类型的 DOM 访问，需要多次访问同一个 DOM 属性或方法需要多次访问时，最好使用一个局部变量缓存此成员。当遍历一个集合时，第一优化原则是把集合存储在局部变量中，并把 `length` 缓存在循环外部，然后，使用局部变量替代这些需要多次读取的元素。

看下面的例子，在循环体中读取元素的三个属性。最慢的版本每次都要读取全局 `document`，优化后的版本缓存了一个集合的引用，最快的版本把当前集合元素存储到一个变量。这三个版本都缓存了集合的 length 属性。

```
// 较慢
function collectionGlobal() {

    var coll = document.getElementsByTagName('div'),
        len = coll.length,
        name = '';
    for (var count = 0; count < len; count++) {
        name = document.getElementsByTagName('div')[count].nodeName;
        name = document.getElementsByTagName('div')[count].nodeType;
        name = document.getElementsByTagName('div')[count].tagName;
    }
    return name;

};

// 较快
function collectionLocal() {

    var coll = document.getElementsByTagName('div'),
        len = coll.length,
        name = '';
    for (var count = 0; count < len; count++) {
        name = coll[count].nodeName;
        name = coll[count].nodeType;
        name = coll[count].tagName;
    }
    return name;

};

// 最快
```

```
function collectionNodesLocal() {

    var coll = document.getElementsByTagName('div'),
        len = coll.length,
        name = '',
        el = null;
    for (var count = 0; count < len; count++) {
        el = coll[count];
        name = el.nodeName;
        name = el.nodeType;
        name = el.tagName;
    }
    return name;

};
```

图 3-5 显示了优化集合循环的好处。第一条柱形图表示通过局部变量引用带来的速度提升，第二条柱形图表示多次读取时缓存集合带来的速度提升。

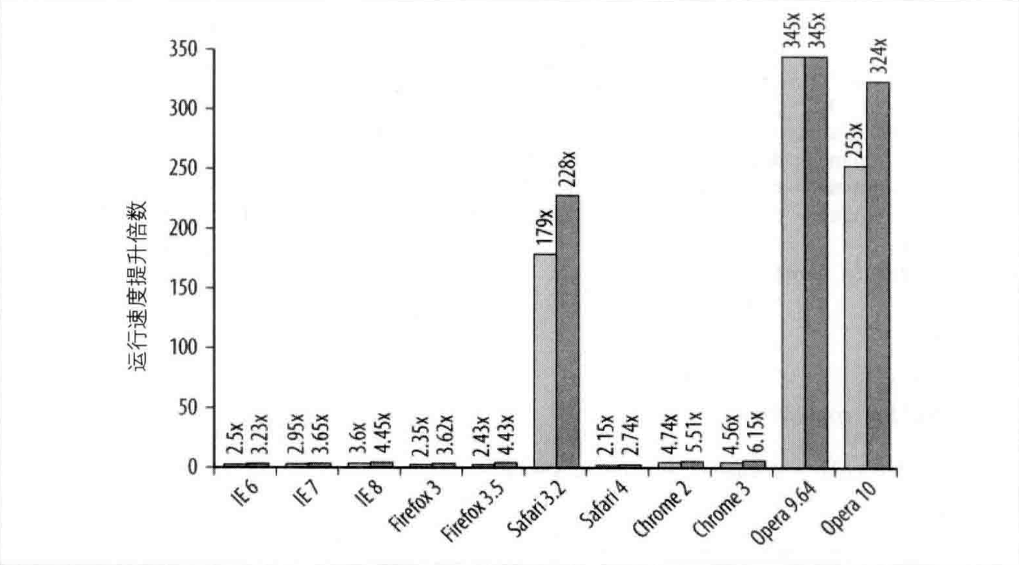


图 3-5　在循环中使用局部变量存储集合引用和集合元素带来的速度提升

遍历 DOM
Walking the DOM

DOM API 提供了多种方法来读取文档结构中的特定部分。当你需要从多种方案中选择时，最好为特定操作选择最高效的 API。

获取 DOM 元素

通常你需要从某一个 DOM 元素开始，操作周围的元素，或者递归查找所有子节点。你可以使用 `childNodes` 得到元素集合，或者用 `nextSibling` 来获取每个相邻元素。

考虑以下两个等价的例子，都是以非递归的方式遍历元素子节点：

```
function testNextSibling() {
    var el = document.getElementById('mydiv'),
        ch = el.firstChild,
        name = '';
    do {
        name = ch.nodeName;
    } while (ch = ch.nextSibling);
    return name;
};

function testChildNodes() {
    var el = document.getElementById('mydiv'),
        ch = el.childNodes,
        len = ch.length,
        name = '';
    for (var count = 0; count < len; count++) {
        name = ch[count].nodeName;
    }
    return name;
};
```

请牢记，`childNodes` 是个元素集合，因此在循环中注意缓存 `length` 属性以避免在每次迭代中更新。

在不同浏览器中，这两种方法的运行时间几乎相等。但是在 IE 中，`nextSibling` 比 `childNode` 表现优异。在 IE 6 中，`nextSibling` 要快 16 倍，IE 7 中是 105 倍。从结果分析，在性能要求极高时，在老版本的 IE 中更推荐使用 `nextSibling` 方法来查找 DOM 节点。其他情况取决于个人或团队偏好。

元素节点

DOM 元素属性诸如 `childNodes`，`firstChild` 和 `nextSibling` 并不区分元素节点和其他类型节点，比如注释和文本节点（通常只是两个节点间的空格）。在某些情况下，只需访问元素节点，因此在循环中很可能需要检查返回节点的类型并过滤掉非元素节点。这些类型检查和过滤其实是不必要的 DOM 操作。

大部分现代浏览器提供的 API 只返回元素节点。如果可用的话推荐使用这些 API，因为它们的执行效率比自己在 JavaScript 代码中实现过滤的效率要高。表 3-1 列出了这些 DOM 属性。

表 3-1　能区分元素节点（HTML 标签）和其他节点的 DOM 属性

属性名	被替代的属性
`children`	`childNodes`
`childElementCount`	`childNodes.length`
`firstElementChild`	`firstChild`
`lastElementChild`	`lastChild`
`nextElementSibling`	`nextSibling`
`previousElementSibling`	`previousSibling`

表 3-1 列出的所有属性都被 Firefox 3.5、Safari 4、Chrome 以及 Opera 9.62 所支持。其中，IE 6、IE 7、IE 8 只支持 children 属性。

使用 `children` 替代 `childNodes` 会更快，因为集合项更少。HTML 源码中的空白实际上是文本节点，而且它并不包含在 `children` 集合中。在所有浏览器中，`children` 都比 `childNodes` 要快，尽管不会快太多——1.5 到 3 倍。值得特别注意的是在 IE 中，遍历 `children` 集合的速度明显快于遍历 `childNode`——IE 6 中要快 24 倍，IE 7 为 124 倍。

选择器 API

对 DOM 中的特定元素操作时，开发者通常需要得到比 `getElementById()`和 `getElementsByTagName()`更好的控制。有时候为了得到需要的元素列表，你需要组合调用它们并遍历返回的节点，但这种繁密的过程效率低下。

另一方面，使用 CSS 选择器也是一种定位节点的便利途径，因为开发者都熟悉 CSS。许多 JavaScript 库为此提供了大量 API，最新的浏览器也提供了一个名为 `querySelectorAll()`的原生 DOM 方法。这种方式自然比使用 JavaScript 和 DOM 来遍历查找元素要快很多。

考虑如下代码：

```
var elements = document.querySelectorAll('#menu a');
```

`elements` 的值包含一个引用列表，指向位于 `id="menu"`的元素之中的所有 a 元素。`querySelectorAll()`方法使用 CSS 选择器作为参数并返回一个 `NodeList`——包含着匹配节点的类数组对象。这个方法不会返回 HTML 集合，因此返回的节点不会对应实时的文档结构。这也避免了本章之前讨论的 HTML 集合引起的性能（和潜在逻辑）问题。

如果不使用 `querySelectorAll()`，为了达到相同的目的你的代码要冗长一些：

```
var elements = document.getElementById('menu').getElementsByTagName ('a');
```

这种情况下，`elements` 会是个 HTML 集合，所以你还需要把它拷贝到数组中，才能得到与 `querySelectorAll()`返回值类似的静态列表。

如果需要处理大量组合查询，使用 `querySelectorAll()`的话会更有效率。比如，页面中有一些 class 为“warning”的 `div` 元素和另一些 class 为“notice”的元素，如果要同时得到它们的列表，建议使用 `querySelectorAll()`：

```
var errs = document.querySelectorAll('div.warning, div.notice');
```

如果不使用 `querySelectorAll()`，要获得相同的结果则复杂得多。其中一种做法是选择所有的 `div` 元素，遍历剔除那些不符合条件的部分。

```
var errs = [],
    divs = document.getElementsByTagName('div'),
    classname = '';
for (var i = 0, len = divs.length; i < len; i++) {
    classname = divs[i].className;
    if (classname === 'notice' || classname === 'warning') {
        errs.push(divs[i]);
    }
}
```

对比这两段代码，显示出使用选择器 API 可带来 2 至 6 倍的速度提升（图 3-6）。

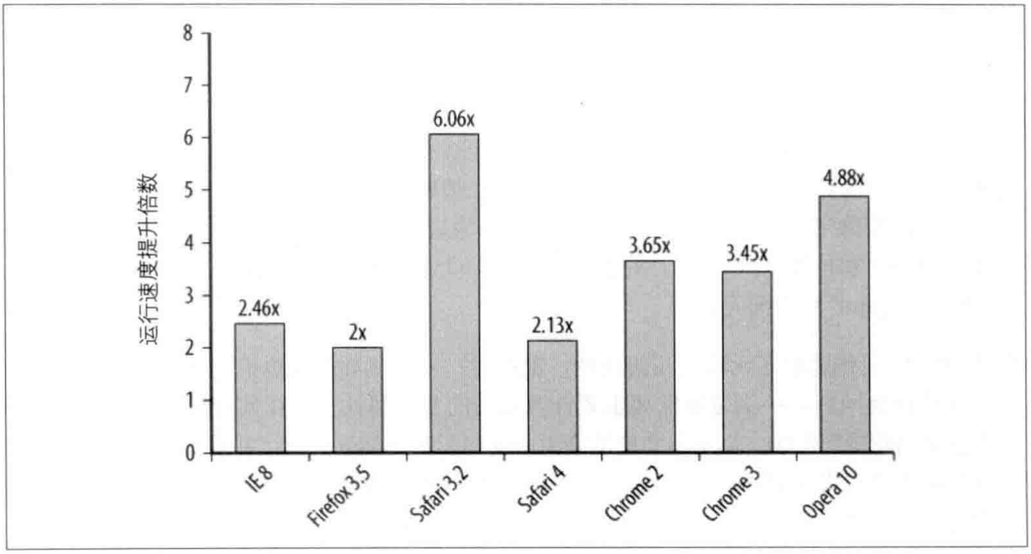


图 3-6　使用选择器 API 替代 getElementsByTagName()带来的速度提升

以下版本的浏览器对选择器 API 提供了原生支持：IE 8、Firefox 3.5、Safari 3.1、Chrome 1 以及 Opera 10。

如图 3-6 所示，选择器 API 的性能更好，所以先检查浏览器是否支持 `document.querySelectorAll()`，如果支持就用。如果你使用 JavaScript 库提供的选择器 API，应确保该库在底层实现中使用原生 API。如果没有的话，也许你该把库升级到新版本了。

你同样还可以使用另一个便利方法——`querySelector()`来获取第一个匹配的节点。

这两个方法都是 DOM 节点的属性，因此可以使用 `document.querySelector('.myclass')`方法来查询整个文档，或通过 `elref.querySelector('.myclass')`在子树中进行查询，这里的 `elref` 是一个 DOM 元素的引用。

重绘与重排

Repaints and Reflows

浏览器下载完页面中的所有组件——HTML 标记、JavaScript、CSS、图片——之后会解析并生成两个内部数据结构：

DOM 树

表示页面结构

渲染树

表示 DOM 节点如何显示

DOM树中的每一个需要显示的节点在渲染树中至少存在一个对应的节点（隐藏的 DOM元素在渲染树中没有对应的节点）。渲染树中的节点被称为“帧（frames）”或“盒（boxes）”，符合 CSS 模型的定义，理解页面元素为一个具有内边距（padding），外边距（margins），边框（borders）和位置（position）的盒子。一旦 DOM 和渲染树构建完成，浏览器就开始显示（绘制“paint”）页面元素。

当 DOM 的变化影响了元素的几何属性（宽和高）——比如改变边框宽度或给段落增加文字，导致行数增加——浏览器需要重新计算元素的几何属性，同样其他元素的几何属性和位置也会因此受到影响。浏览器会使渲染树中受到影响的部分失效，并重新构造渲染树。这个过程称为“重排（reflow）”。完成重排后，浏览器会重新绘制受影响的部分到屏幕中，该过程称为“重绘（repaint）”。

并不是所有的 DOM 变化都会影响几何属性。例如，改变一个元素的背景色并不会影响它的宽和高。在这种情况下，只会发生一次重绘（不需要重排），因为元素的布局并没有改变。

重绘和重排操作都是代价昂贵的操作，它们会导致 Web 应用程序的 UI 反应迟钝。所以，应当尽可能减少这类过程的发生。

重排何时发生
When Does a Reflow Happen

正如前文所提到的，当页面布局和几何属性改变时就需要“重排”。下述情况中会发生重排。

- 添加或删除可见的 DOM 元素。
- 元素位置改变。
- 元素尺寸改变（包括：外边距、内边距、边框厚度、宽度、高度等属性改变）。
- 内容改变，例如：文本改变或图片被另一个不同尺寸的图片替代。
- 页面渲染器初始化。
- 浏览器窗口尺寸改变。

根据改变的范围和程度，渲染树中或大或小的对应的部分也需要重新计算。有些改变会触发整个页面的重排：例如，当滚动条出现时。

渲染树变化的排队与刷新
Queuing and Flushing Render Tree Changes

由于每次重排都会产生计算消耗，大多数浏览器通过队列化修改并批量执行来优化重排过程。然而，你可能会（经常不知不觉）强制刷新队列并要求计划任务立刻执行。获取布局信息的操作会导致列队刷新，比如以下方法：

- `offsetTop, offsetLeft, offsetWidth, offsetHeight`
- `scrollTop, scrollLeft, scrollWidth, scrollHeight`
- `clientTop, clientLeft, clientWidth, clientHeight`
- `getComputedStyle() (currentStyle in IE)`

以上属性和方法需要返回最新的布局信息，因此浏览器不得不执行渲染列队中的“待处理变化”并触发重排以返回正确的值。

在修改样式的过程中，最好避免使用上面列出的属性。它们都会刷新渲染队列，即使你是在获取最近未发生改变的或者与最新改变无关的布局信息。

考虑下面的例子，它三次改变同一个样式属性（这种情况可能不会出现在真实的代码中，不过它能独立展示一个重要话题）：

```
// 定义变量并获取样式
var computed,
    tmp = '',
    bodystyle = document.body.style;

if (document.body.currentStyle) { // IE, Opera
    computed = document.body.currentStyle;
} else { // W3C
    computed = document.defaultView.getComputedStyle(document.body, '');
}

// 修改同一属性低效的方式
// 然后获取样式信息
bodystyle.color = 'red';
tmp = computed.backgroundColor;
bodystyle.color = 'white';
tmp = computed.backgroundImage;
bodystyle.color = 'green';
tmp = computed.backgroundAttachment;
```

示例中，`body` 元素的前景色被修改了三次，每次修改后都读取一个 computed 样式属性。读取的属性——`backgroundColor`、`backgroundImage`、`backgroundAttachment` 都与改变的颜色无关。然而浏览器却需要刷新渲染队列并重排，因为 computed 的样式属性被请求了。

一个更有效率的方法是不要在布局信息改变时查询它。如果读取 computed 样式的代码被移到末尾，代码看起来如下：

```
bodystyle.color = 'red';
bodystyle.color = 'white';
bodystyle.color = 'green';
tmp = computed.backgroundColor;
tmp = computed.backgroundImage;
tmp = computed.backgroundAttachment;
```

在所有浏览器中，该方法都更快，参见图 3-7。

最小化重绘和重排
Minimizing Repaints and Reflows

重绘和重排可能代价非常昂贵，因此一个好的提高程序响应速度的策略就是减少此类操作的发生。为了减少发生次数，应该合并多次对 DOM 和样式的修改，然后一次处理掉。

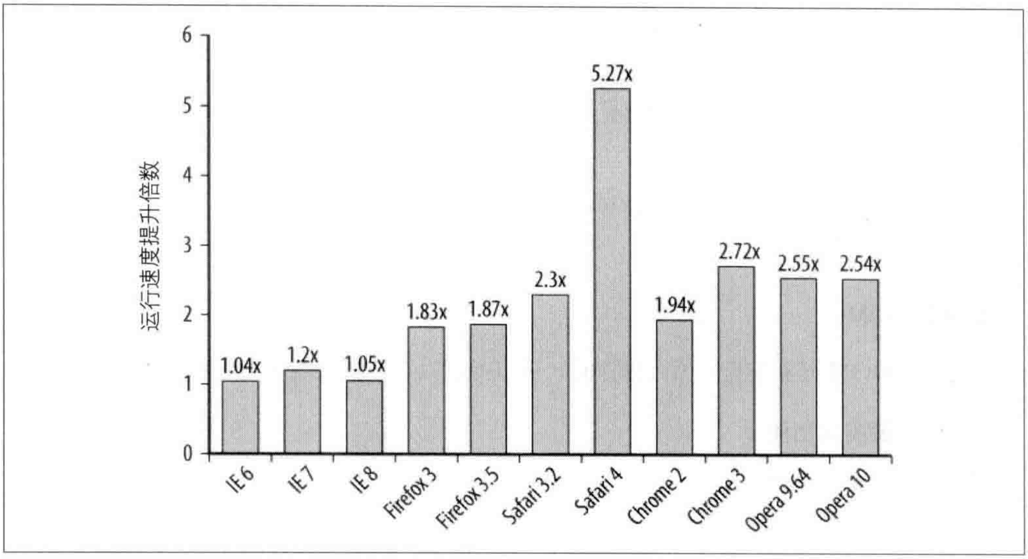


图 3-7　通过延迟访问布局信息来避免“重排”所带来的性能提升

改变样式

考虑这个例子：

```
var el = document.getElementById('mydiv');
el.style.borderLeft = '1px';
el.style.borderRight = '2px';
el.style.padding = '5px';
```

示例中有三个样式属性被改变，每一个都会影响元素的几何结构。最糟糕的情况下，会导致浏览器触发三次重排。大部分现代浏览器为此做了优化，只会触发一次重排，但是在旧版浏览器中或者有一个分离的异步处理过程时（比如使用计时器），仍然效率低下。如果在上面代码执行时，有其他代码请求布局信息，这会导致触发三次重排。而且，这段代码四次访问 DOM，可以被优化。

一个能够达到同样效果且效率更高的方式是：合并所有的改变然后一次处理，这样只会修改 DOM 一次。使用 cssText 属性可以实现：

```
var el = document.getElementById('mydiv');
el.style.cssText = 'border-left: 1px; border-right: 2px; padding: 5px;';
```

例子中的代码修改 cssText 属性并覆盖了已存在的样式信息，因此如果想保留现有样式，可以把它附加在 cssText 字符串后面。

```
el.style.cssText += '; border-left: 1px;';
```

另一个一次性修改样式的办法是修改 CSS 的 class 名称，而不是修改内联样式。这种方法适用于那些不依赖于运行逻辑和计算的情况。改变 CSS 的 class 名称的方法更清晰，更易于维护；它有助于保持你的脚本与免除显示性代码，尽管它可能带来轻微的性能影响，因为改变类时需要检查级联样式。

```
var el = document.getElementById('mydiv');
el.className = 'active';
```

批量修改 DOM

当你需要对 DOM 元素进行一系列操作时，可以通过以下步骤来减少重绘和重排的次数：

1. 使元素脱离文档流。
2. 对其应用多重改变。
3. 把元素带回文档中。

该过程里会触发两次重排——第一步和第三步。如果你忽略这两个步骤，那么在第二步所产生的任何修改都会触发一次重排。

有三种基本方法可以使 DOM 脱离文档：

- 隐藏元素，应用修改，重新显示。
- 使用文档片断（docuement fragment）在当前 DOM 之外构建一个子树，再把它拷贝回文档。
- 将原始元素拷贝到一个脱离文档的节点中，修改副本，完成后再替换原始元素。

为了演示脱离文档的操作，考虑下面的链接列表，它必须更新更多信息：

```
<ul id="mylist">
   <li><a href="http://phpied.com">Stoyan</a></li>
   <li><a href="http://julienlecomte.com">Julien</a></li>
</ul>
```

假设附加数据已经存储在一个对象中，并要插入列表。这些数据定义如下：

```
var data = [
  {
    "name": "Nicholas",
    "url": "http://nczonline.net"
  },
  {
    "name": "Ross",
    "url": "http://techfoolery.com"
  }
];
```

下面是一个用来更新指定节点数据的通用函数：

```
function appendDataToElement(appendToElement, data) {
    var a, li;
    for (var i = 0, max = data.length; i < max; i++) {
        a = document.createElement('a');
        a.href = data[i].url;
        a.appendChild(document.createTextNode(data[i].name));
        li = document.createElement('li');
        li.appendChild(a);
        appendToElement.appendChild(li);
    }
};
```

更新列表内容而不用担心重排问题，最明显的方法如下：

```
var ul = document.getElementById('mylist');
appendDataToElement(ul, data);
```

然而，使用这种方法，`data` 数组的每一个新条目被附加到当前 DOM 树时都会导致重排。正如前文所讨论的那样，一个减少重排的方法是通过改变 `display` 属性，临时从文档中移除 `<ul>`元素，然后再恢复它：

```
var ul = document.getElementById('mylist');
ul.style.display = 'none';
appendDataToElement(ul, data);
ul.style.display = 'block';
```

另一种减少重排次数的办法是：在文档之外创建并更新一个文档片断，然后把它附加到原始列表中。文档片断是个轻量级的 `document` 对象，它的设计初衷就是为了完成这类任务——更新和移动节点。文档片断的一个便利的语法特性是当你附加一个片断到节点中时，实际上被添加的是该片断的子结点，而不是片断本身。下面的例子少一行代码，只触发一次重排，而且只访问了一次实时的 DOM：

```
var fragment = document.createDocumentFragment();
appendDataToElement(fragment, data);
document.getElementById('mylist').appendChild(fragment);
```

第三种解决方案是为需要修改的节点创建一个备份，然后对副本进行操作，一旦操作完成，就用新的节点替代旧的节点。

```
var old = document.getElementById('mylist');
var clone = old.cloneNode(true);
appendDataToElement(clone, data);
old.parentNode.replaceChild(clone, old);
```

推荐尽可能地使用文档片断（第二个方案），因为它们所产生的 DOM 遍历和重排次数最少。唯一潜在的问题是文档片断未被充分利用，有些团队成员可能并不熟悉这项技术。

缓存布局信息

Caching Layout Information

如前文所述，浏览器尝试通过队列化修改和批量执行的方式最小化重排次数。当你查询布局信息时，比如获取偏移量（offsets）、滚动位置（scroll values）或计算出的样式值（computedsytle values）时，浏览器为了返回最新值，会刷新队列并应用所有变更。最好的做法是尽量减少布局信息的获取次数，获取后把它赋值给局部变量，然后再操作局部变量。

考虑一个例子，把 `myElement` 元素沿对角线移动，每次移动一个像素，从 100 像素 × 100 像素的位置开始，到 500 像素 × 500 像素的位置结束。在 timeout 循环体中你可以使用下面的方法：

```
// 低效的
myElement.style.left = 1 + myElement.offsetLeft + 'px';
myElement.style.top = 1 + myElement.offsetTop + 'px';
if (myElement.offsetLeft >= 500) {
    stopAnimation();
}
```

这种方法效率低下，因为元素每次移动时都会查询偏移量，导致浏览器刷新渲染列队而不利于优化。一个更好的方法是，获取一次起始位置的值，然后将其赋值给一个变量，比如 `var current = myElement.offsetLeft`。然后，在动画循环中，直接使用 `current` 变量而不再查询偏移量：

```
current++
myElement.style.left = current + 'px';
myElement.style.top = current + 'px';
if (current >= 500) {
    stopAnimation();
}
```

让元素脱离动画流

Take Elements Out of the Flow for Animations

用展开/折叠的方式来显示和隐藏部分页面是一种常见的交互模式。它通常包括展开区域的几何动画，并将页面其他部分推向下方。

一般来说，重排只影响渲染树中的一小部分，但也可能影响很大的部分，甚至整个渲染树。浏览器所需要重排的次数越少，应用程序的响应速度就越快。因此当页面顶部的一个动画推移页面整个余下的部分时，会导致一次代价昂贵的大规模重排，让用户感到页面一顿一顿的。渲染树中需要重新计算的节点越多，情况就会越糟。

使用以下步骤可以避免页面中的大部分重排：

1. 使用绝对位置定位页面上的动画元素，将其脱离文档流。
2. 让元素动起来。当它扩大时，会临时覆盖部分页面。但这只是页面一个小区域的重绘过程，不会产生重排并重绘页面的大部分内容。
3. 当动画结束时恢复定位，从而只会下移一次文档的其他元素。

IE 和:hover

IE and :hover

从 IE 7 开始，IE 允许在任何元素（严格模式下）上使用`:hover`这个 CSS 伪选择器。然而，如果你有大量元素使用了`:hover`，那么会降低响应速度。此问题在 IE 8 中更为明显。

例如，如果你创建一个 5 列和 500~1000 行的表格，并使用 `tr:hover` 改变背景色来高亮显示鼠标所在的当前行，当鼠标在表格上移动时，性能会降低。高亮过程会变慢，CPU 使用率会提高到 80%~90%。所以在元素很多时应避免使用这种效果，比如很大的表格或很长的列表。

事件委托

Event Delegation

当页面中存在大量元素，而且每一个都要一次或多次绑定事件处理器（比如 `onclick`）时，这种情况可能会影响性能。每绑定一个事件处理器都是有代价的，它要么是加重了页面负担（更多的标签或 JavaScript 代码），要么是增加了运行期的执行时间。需要访问和修改的 DOM 元素越多，应用程序也就越慢，特别是事件绑定通常发生在`onload`（或 `DOMContentReady`）时，此时对每一个富交互应用的网页来说都是一个拥堵的时刻。事件绑定占用了处理时间，而且，浏览器需要跟踪每个事件处理器，这也会占用更多的内存。当这些工作结束时，这些事件处理器中的绝大部分都不再需要（因为并不是 100%的按钮或链接会被用户点击），因此有很多工作是没必要的。

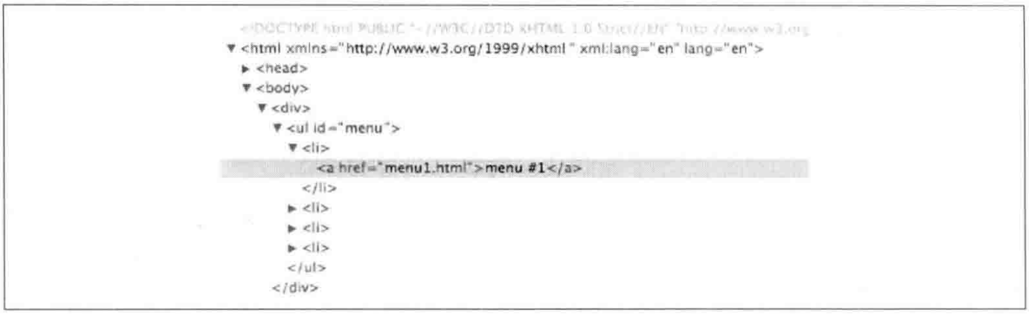
一个简单而优雅的处理 DOM 事件的技术是事件委托。它是基于这样一个事实：事件逐层冒泡并能被父级元素捕获。使用事件代理，只需给外层元素绑定一个处理器，就可以处理在其子元素上触发的所有事件。

根据 DOM 标准，每个事件都要经历三个阶段：

- 捕获
- 到达目标

- 冒泡

IE 不支持捕获，但对于委托而言，冒泡已经足够。考虑图 3-8 所示的页面结构。



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org
▼ <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  ▶ <head>
  ▼ <body>
    ▼ <div>
      ▼ <ul id="menu">
        ▼ <li>
            <a href="menu1.html">menu #1</a>
          </li>
        ▶ <li>
        ▶ <li>
        ▶ <li>
        </ul>
      </div>
```

图 3-8　一个 DOM 树的例子

当用户点击链接“menu #1”，点击事件首先由`<a>`元素收到，然后向 DOM 树上层冒泡，被`<li>`元素接收，接着是`<ul>`，然后是`<div>`，以此类推，一直到达 document 的顶层乃至 `window`。这使得你可以添加一个事件处理器到父级元素，由它接收所有子节点的事件消息。

也许你想为图中的文档提供一个渐进增强的 Ajax 体验。如果用户浏览器禁用了 JavaScript，点击菜单中的链接仍然可以重载页面。但是如果开启了 JavaScript 且客户端功能足够强大，你想要拦截所有点击事件，并阻止其默认行为（打开链接），发送一个 Ajax 请求来获取内容，然后局部更新页面。用事件委托来实现这个过程，你只需要给所有链接的外层 UL “menu”元素添加一个点击监听器，它会捕获并分析点击是否来自链接。

```
document.getElementById('menu').onclick = function(e) {

    // 浏览器 target
    e = e || window.event;
    var target = e.target || e.srcElement;

    var pageid, hrefparts;

    // 只关心 hrefs，非链接点击则退出
    if (target.nodeName !== 'A') {
        return;
    }

    // 从链接中找出页面 ID
    hrefparts = target.href.split('/');
    pageid = hrefparts[hrefparts.length - 1];
```

```
        pageid = pageid.replace('.html', '');

        // 更新页面
        ajaxRequest('xhr.php?page=' + id, updatePageContents);

        // 浏览器组织默认行为并取消冒泡
        if (typeof e.preventDefault === 'function') {
            e.preventDefault();
            e.stopPropagation();
        } else {
            e.returnValue = false;
            e.cancelBubble = true;
        }

    };
```

正如你所看到的那样，事件委托技术并不复杂；你只需检查事件是否来自你所预期的元素。尽管有一些冗长的浏览器兼容性代码，但如果你把这部分移入可重用的类库，代码就会变得相当干净。跨浏览器兼容的部分包括：

- 访问事件对象，并判断事件源。
- 取消文档树中的冒泡（可选）。
- 阻止默认动作（可选，但本例需要，因为需要捕获并阻止打开链接）。

小结
Summary

访问和操作 DOM 是现代 Web 应用的重要部分。但每次穿越连接 ECMAScript 和 DOM 两个岛屿之间的桥梁，都会被收取“过桥费”。为了减少 DOM 编程带来的性能损失，请记住以下几点：

- 最小化 DOM 访问次数，尽可能在 JavaScript 端处理。
- 如果需要多次访问某个 DOM 节点，请使用局部变量存储它的引用。
- 小心处理 HTML 集合，因为它实时连系着底层文档。把集合的长度缓存到一个变量中，并在迭代中使用它。如果需要经常操作集合，建议把它拷贝到一个数组中。
- 如果可能的话，使用速度更快的 API，比如 `querySelectorAll()`和 `firstElementChild`。
- 要留意重绘和重排；批量修改样式时，“离线”操作 DOM 树，使用缓存，并减少访问布局信息的次数。
- 动画中使用绝对定位，使用拖放代理。
- 使用事件委托来减少事件处理器的数量。

第 4 章

算法和流程控制

Algorithms and Flow Control

代码的整体结构是影响运行速度的主要因素之一。代码数量少并不意味着运行速度就快，代码数量多也不意味着运行速度一定慢。代码的组织结构和解决具体问题的思路是影响代码性能的主要因素。

本章讨论的技术并不限于 JavaScript，同样适用于其他语言的性能优化。尽管有些来自其他语言的优化建议可能与 JavaScript 存在差异，但多种多样的 JavaScript 引擎以及它们各自的奇技淫巧仍然是重点讨论对象，所有的技术都是源于现代计算机科学知识。

循环

Loops

在大多数编程语言中，代码执行时间大部分消耗在循环中。循环处理是最常见的编程模式之一，也是提升性能必须关注的要点之一。理解 JavaScript 中循环对性能的影响至关重要，因为死循环或长时间运行的循环会严重影响用户体验。

循环的类型

Types of Loops

ECMA-262 标准第三版定义了 JavaScript 的基本语法和行为，其中有四种循环类型。第一种是标准 `for` 循环，与其他类 C 语言的语法相同：

```
for (var i=0; i < 10; i++){
    // 循环主体
}
```

`for` 循环是 JavaScript 中最常用的循环结构。它由四部分组成：初始化、前测条件、后执行体、循环体。当代码运行中遇到 `for` 循环时，先运行初始化代码，然后进入前测条件。如果前测条件的结果为 `true`，则运行循环体。循环体执行完后，后执行代码开始运行。`for` 循环直观的代码封装风格被开发者们所喜爱。

提示：请注意，在 `for` 循环初始化中的 `var` 语句会创建一个函数级的变量，而不是循环级。由于 JavaScript 只有函数级作用域，因此在 `for` 循环中定义一个新变量相当于在循环体外定义一个新变量。

第二种循环类型是 `while` 循环。`while` 循环是最简单的前测循环，由一个前测条件和一个循环体构成：

```
var i = 0;
while(i < 10){
    // 循环主体
    i++;
}
```

在循环体运行前，先计算前测条件。如果计算结果为`true`，就运行循环体；否则，循环体会被跳过。任何 `for` 循环都能改写成 `while` 循环，反之亦然。

第三种循环类型是 `do-while` 循环。`do-while` 循环是 JavaScript 中唯一一种后测循环，它由两部分组成，循环体和后测条件：

```
var i = 0;
do {
   // 循环主体
} while (i++ < 10);
```

在 `do-while` 循环中，循环体会至少运行一次，而后再由后测条件决定是否再次运行。

第四种也是最后一种循环类型是 `for-in` 循环。该循环有个非常特别的用途：它可以枚举任何对象的属性名。基本格式如下：

```
for (var prop in object){
    // 循环主体
}
```

循环体每次运行时，`prop` 变量被赋值为 `object` 的一个属性名（字符串），直到所有属性遍历完成才返回。所返回的属性包括对象实例属性以及从原型链中继承而来的属性。

循环性能

Loop Performance

不断引发循环性能争论的源头是循环类型的选择。在 JavaScript 提供的四种循环类型中，只有 `for-in` 循环比其他几种明显要慢。

由于每次迭代操作会同时搜索实例或原型属性，`for-in`循环的每次迭代都会产生更多开销，所以比其他循环类型要慢。对比相同迭代次数的循环，`for-in`循环最终只有其他类型速度的1/7。因此，除非你明确需要迭代一个属性数量未知的对象，否则应避免使用`for-in` 循环。如果你需要遍历一个数量有限的已知属性列表，使用其他循环类型会更快，比如可以使用以下模式：

```
var props = ["prop1", "prop2"],
    i = 0;

while (i < props.length){
    process(object[props[i++]]);
}
```

这段代码创建了一个由属性名构成的数组。`while` 循环用来遍历这个属性列表并处理对应的对象成员。相对于查找该对象的每一个属性，该代码只关注给定的属性，减少了循环的开销，也节省了时间。

提示：不要使用 `for-in` 来遍历数组成员。

除 `for-in` 循环外，其他循环类型的性能都差不多，深究哪种循环最快没有什么意义。循环类型的选择应该基于需求而不是性能。

如果循环类型与性能无关，那么该如何选择？其实只有两个可选因素：

- 每次迭代处理的事务。
- 迭代的次数。

通过减少这两者中的一个或者全部的时间开销，你就提升循环的整体性能。

减少迭代的工作量

很明显，如果一次循环迭代要花很长时间去执行，那么多次循环意味着需要更多时间。一个提升循环整体速度的好方法是限制循环中耗时操作的数量。

一个典型的数组处理循环可以采用三种循环中的任何一种。最常见的写法如下：

```
// 原始版本
for (var i=0; i < items.length; i++){
    process(items[i]);
}

var j=0;
while (j < items.length){
    process(items[j++]);
}

var k=0;
do {
    process(items[k++]);
} while (k < items.length);
```

在上面的循环中，每次运行循环体时都会产生如下操作：

1. 在控制条件中查找一次属性（`items.length`）。
2. 在控制条件中执行一次数值比较（`i < items.length`）。
3. 一次比较操作查看控制条件的计算结果是否为 true（`i < items.length == true`）。
4. 一次自增操作（`i++`）。
5. 一次数组查找（`items[i]`）。
6. 一次函数调用（`process(items[i])`）。

在这些简单的循环中，即使代码不多，每次迭代也要进行许多操作。代码运行速度很大程度上取决于函数 `process()`对每个数组项的操作，即使如此，减少每次迭代中的操作总数能大幅提高循环的总体性能。

优化循环的第一步是要减少对象成员及数组项的查找次数。正如第 2 章所讨论的，在大多数浏览器中，这些操作比使用局部变量和字面量需要花更多时间。前面的例子中每次循环都要查找 `items.length`，这样做很耗时，由于该值在循环运行过程中并没改变，因此产生了不必要的性能损失。提高这个循环的性能很简单，只查找一次属性，并把值存储到一个局部变量，然后在控制条件中使用这个变量：

```
// 最小化属性查找
for (var i=0, len=items.length; i < len; i++){
    process(items[i]);
}

var j=0,
    count = items.length;
```

```
while (j < count){
    process(items[j++]);
}

var k=0,
    num = items.length;
do {
    process(items[k++]);
} while (k < num);
```

这些重写后的循环只在循环运行前对数组长度进行一次属性查找。这使得控制条件可直接读取局部变量，所以速度更快。根据数组的长度，在大多数浏览器中能节省大概 25%的运行时间（IE 中甚至可以节省 50%）。

你还可以通过颠倒数组的顺序来提高循环性能。通常，数组项的顺序与所要执行的任务无关，因此从最后一项开始向前处理是个备选方案。倒序循环是编程语言中一种通用的性能优化方法，但一般来说不是那么容易理解。在 JavaScript 中，倒序循环会略微提升性能，前提是你排除那些额外操作带来的影响：

```
//减少属性查找并反转
for (var i=items.length; i--; ){
    process(items[i]);
}

var j = items.length;
while (j--){
    process(items[j]);
}

var k = items.length-1;
do {
    process(items[k]);
} while (k--);
```

本例中使用了倒序循环，并把减法操作放在控制条件中。现在每个控制条件只是简单地与零比较。控制条件与 `true` 值比较时，任何非零数会自动转换为 `true`，而零值等同于 `false`。实际上，控制条件已经从两次比较（迭代数少于总数吗？它是否为 `true`？）减少到一次比较（它是 `true` 吗？）。每次迭代从两次比较减少到一次，进一步提高了循环速度。通过倒序循环和减少属性查找，你可以看到运行速度比原始版本快了 50%～60%。

对比原始版本，每次迭代中只有如下操作：

1. 一次控制条件中的比较（`i == true`）。
2. 一次减法操作（`i--`）。
3. 一次数组查找（`items[i]`）。
4. 一次函数调用（`process(items[i])`）。

新的循环代码在每次迭代中减少了两次操作，随着迭代次数增加，性能的提升会更趋明显。

提示：当循环复杂度为O(n)时，减少每次迭代的工作量是最有效的方法。当复杂度大于 O(n)，建议着重减少迭代次数。

减少迭代次数

即使是循环体中执行最快的代码，累计迭代上千次也会变慢下来。此外，循环体运行时会带来一次小的性能开销，这增加了总体运行时间。减少迭代次数能获得更加显著的性能提升。最广为人知的一种限制循环迭代次数的模式被成为“达夫设备（Duff’s Device）”[译注1]。

“Duff’s Device”是一个循环体展开技术，它使得一次迭代中实际上执行了多次迭代的操作。Jeff Greenberg 被认为是将“Duff’s Device”代码从原始的 C 实现移植到 JavaScript 中的第一人。一个典型实现如下：

```
//credit: Jeff Greenberg
var iterations = Math.floor(items.length / 8),
    startAt    = items.length % 8,
    i          = 0;

do {
    switch(startAt){
        case 0: process(items[i++]);
        case 7: process(items[i++]);
        case 6: process(items[i++]);
        case 5: process(items[i++]);
        case 4: process(items[i++]);
        case 3: process(items[i++]);
        case 2: process(items[i++]);
        case 1: process(items[i++]);
    }
    startAt = 0;
} while (--iterations);
```

Duff’s Device 背后的基本理念是：每次循环中最多可调用 8 次 `process()`。循环的迭代次数为总数除以 8。由于不是所有数字都能被 8 整除，变量 `startAt` 用来存放余数，表示第一次循环

[译注1] Duff’s_device，参见 *http://en.wikipedia.org/wiki/Duff’s_device*。

中应调用多少次 `process()`。如果是 12 次，那么第一次循环会调用 `process()` 4 次，第二次循环调用 `process()` 8 次，用两次循环替代了 12 次循环。

此算法一个稍快的版本取消了 `switch` 语句，并将余数处理和主循环分开：

```
//credit: Jeff Greenberg
var i = items.length % 8;
while(i){
    process(items[i--]);
}

i = Math.floor(items.length / 8);

while(i){
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
}
```

尽管这种实现方式用两次循环代替之前的一次循环，但它移除了循环体中的 `switch` 语句，速度比原始循环更快。

是否应该使用 Duff's Device，无论是原始版本还是修改后的版本，都很大程度上依赖于迭代次数。如果循环迭代次数小于 1000，你很有可能看到它与常规循环结构相比只有微不足道的性能提升。如果迭代数超过 1000，那么 Duff's Device 的执行效率将明显提升。例如在 500 000 次迭代中，其运行时间比常规循环少 70%。

基于函数的迭代

Function-Based Iteration

ECMA-262 标准第四版引入了一个新的原生数组方法：`forEach()`。此方法遍历一个数组的所有成员，并在每个成员上执行一个函数。要运行的函数作为参数传给 `forEach()`，并在调用时接收三个参数，分别是：当前数组项的值、索引以及数组本身。以下是个使用示例：

```
items.forEach(function(value, index, array){
    process(value);
});
```

在 Firefox、Chrome 和 Safari 中，`forEach()`被原生支持。此外，大多数 JavaScript 类库都有等价实现：

```
//YUI 3
Y.Array.each(items, function(value, index, array){
    process(value);
});

//jQuery
jQuery.each(items, function(index, value){
    process(value);
});

//Dojo
dojo.forEach(items, function(value, index, array){
    process(value);
});

//Prototype
items.each(function(value, index){
    process(value);
});

//MooTools
$each(items, function(value, index){
    process(value);
});
```

尽管基于函数的迭代提供了一个更为便利的迭代方法，但它仍然比基于循环的迭代要慢一些。对每个数组项调用外部方法所带来的开销是速度慢的主要原因。在所有情况下，基于循环的迭代比基于函数的迭代快 8 倍，因此在运行速度要求严格时，基于函数的迭代不是合适的选择。

条件语句

Conditionals

与循环的原理类似，条件表达式决定了 JavaScript 运行流的走向。其他语言对应该使用 `if-else` 语句还是 `switch` 语句的传统观点同样适用于 JavaScript。由于不同的浏览器针对流程控制进行了不同的优化，因此使用哪种技术更好没有定论。

if-else 对比 switch

if-else Versus switch

使用 `if-else` 还是 `switch`，最流行的方法是基于测试条件的数量来判断：条件数量越大，越倾向于使用 `switch` 而不是 `if-else`。这通常归结于代码的易读性。这个观点认为，当循环条件

较少时 `if-else` 更易读，当条件数量较多时 `switch` 更易读。考虑如下代码：

```
if (found){
    // 代码处理
} else {
    // 其他代码处理
}

switch(found){
    case true:
        // 代码处理
        break;

    default:
        // 其他代码处理
}
```

尽管两块代码完成的是相同的任务，但还是有很多人会认为 `if-else` 语句比 `switch` 更易读。然而，如果增加条件语句的数量，这种观点会被扭转过来：

```
if (color == "red"){
    // 代码处理
} else if (color == "blue"){
    // 代码处理
} else if (color == "brown"){
    // 代码处理
} else if (color == "black"){
    // 代码处理
} else {
    // 代码处理
}

switch (color){
    case "red":
        // 代码处理
        break;

    case "blue":
        // 代码处理
        break;

    case "brown":
        // 代码处理
        break;

    case "black":
        // 代码处理
        break;

    default:
        // 代码处理
}
```

大多数人会认为这段代码中的 `switch` 表达式比 `if-else` 表达式可读性更好。

事实证明，大多数情况下 `switch` 比 `if-else` 运行得要快，但只有当条件数量很大时才快得明显。这两个语句主要性能区别是：当条件增加时，`if-else` 性能负担增加的程度比 `switch` 要多[译注2]。因此，我们自然倾向于在条件数量较少时使用 `if-else`，而在条件数量较大时使用 `switch`，这从性能方面考虑也是合理的。

通常来说，`if-else` 适用于判断两个离散值或几个不同的值域。当判断多于两个离散值时，`switch` 语句是更佳选择。

优化 if-else
Optimizing if-else

优化 `if-else` 的目标是：最小化到达正确分支前所需判断的条件数量。最简单的优化方法是确保最可能出现的条件放在首位。考虑如下代码：

```
if (value < 5) {
    // 代码处理
} else if (value > 5 && value < 10) {
    // 代码处理
} else {
    // 代码处理
}
```

该代码只有当 `value` 值经常小于 5 的时候才是最优的。如果 `value` 大于 5 或者等于 10，那么每次到达正确分支之前必须经过两个条件判断，最终增加了这个语句所消耗的平均时间。`if-else` 中的条件语句应该总是按照从最大概率到最小概率的顺序排列，以确保运行速度最快。

另一种减少条件判断次数的方法是把 `if-else` 组织成一系列嵌套的 `if-else` 语句。使用单个庞大的 `if-else` 通常会导致运行缓慢，因为每个条件都需要判断。例如：

```
if (value == 0){
    return result0;
} else if (value == 1){
    return result1;
} else if (value == 2){
    return result2;
} else if (value == 3){
    return result3;
} else if (value == 4){
    return result4;
} else if (value == 5){
    return result5;
} else if (value == 6){
```

[译注2] 大多数的语言对 switch 语句的实现都采用了 branch table（分支表）索引来进行优化，更多内容参考：*http://en.wikipedia.org/wiki/Switch_statement*。另外，在 JavaScript 中，switch 语句比较值时使用全等操作符，不会发生类型转换的损耗。

```
    return result6;
} else if (value == 7){
    return result7;
} else if (value == 8){
    return result8;
} else if (value == 9){
    return result9;
} else {
    return result10;
}
```

在这个 `if-else` 表达式中，条件语句最多要判断 10 次。假设 `value` 的值在 0 到 10 之间均匀分布，那么这会增加平均运行时间。为了最小化条件判断的次数，代码可重写为一系列嵌套的 `if-else` 语句，比如：

```
if (value < 6){

    if (value < 3){
        if (value == 0){
            return result0;
        } else if (value == 1){
            return result1;
        } else {
            return result2;
        }
    } else {
        if (value == 3){
            return result3;
        } else if (value == 4){
            return result4;
        } else {
            return result5;
        }
    }

} else {

    if (value < 8){
        if (value == 6){
            return result6;
        } else {
            return result7;
        }
    } else {
        if (value == 8){
            return result8;
        } else if (value == 9){
            return result9;
        } else {
            return result10;
        }
    }
}
```

重写后的 `if-else` 语句每次到达正确分支时最多经过 4 次条件判断。它使用二分法把值域分成一系列的区间，然后逐步缩小范围。当值的范围均匀分布在 0 到 10 之间时，代码运行的平均时间大约是前面例子的一半。这个方法非常适用于有多个值域需要测试的时候（如果是离散值，那么 `switch` 语句通常更为合适）。

查找表[译注3]

Lookup Tables

有些时候优化条件语句的最佳方案是避免使用 `if-else` 和 `switch`。当有大量离散值需要测试时，`if-else` 和 `switch` 比使用查找表慢很多。JavaScript 中可以使用数组和普通对象来构建查找表，通过查找表访问数据比用 `if-else` 或 `switch` 快很多，特别是在条件语句数量很大的时候（图 4-1）。

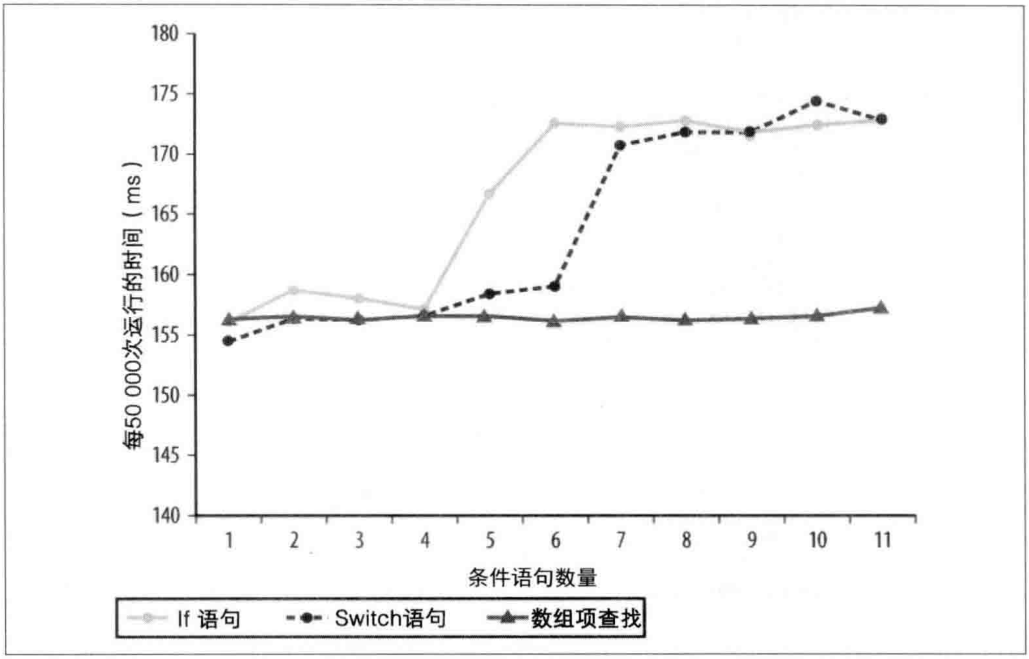


图 4-1　IE 7 中用数组项查找与 if-else 或 switch 的比较

使用查找表相对于 `if-else` 和 `switch`，不仅速度更快，而且有时代码的可读性更好，特别是当需要测试的离散值数量非常大的时候，在那种情况下，`switch` 语句变得很笨重：

译注3：　LookupTable 是计算机专业术语，参见 *http://zh.wikipedia.org/zh-cn/查找表*。

```
switch(value){
    case 0:
        return result0;
    case 1:
        return result1;
    case 2:
        return result2;
    case 3:
        return result3;
    case 4:
        return result4;
    case 5:
        return result5;
    case 6:
        return result6;
    case 7:
        return result7;
    case 8:
        return result8;
    case 9:
        return result9;
    default:
        return result10;
}
```

`switch` 表达式代码所占的空间可能与它的重要性不成比例。整个结构可以用一个数组作为查找表来替代：

```
// 将返回值集合存入数组
var results = [result0, result1, result2, result3, result4, result5, result6,
               result7, result8, result9, result10]

// 返回当前结果
return results[value];
```

当你使用查找表时，必须完全抛弃条件判断语句。这个过程变成数组项查询或者对象成员查询。查找表的一个主要优点是：不用书写任何条件判断语句，即便候选值数量增加时，也几乎不会产生额外的性能开销。

当单个键和单个值之间存在逻辑映射时（正如前面的例子），查找表的优势就能体现出来。`switch` 语句更适合于每个键都需要对应一个独特的动作或一系列动作的场合。

递归

Recursion

使用递归可以把复杂的算法变得简单。事实上有些传统算法正是用递归实现的，比如阶乘函数：

```
function factorial(n){
    if (n == 0){
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

递归函数的潜在问题是终止条件不明确或缺少终止条件会导致函数长时间运行，并使得用户界面处于假死状态。而且，递归函数还可能遇到浏览器的“调用栈大小限制”（Call stack size limites）。

调用栈限制

Call Stack Limits

JavaScript 引擎支持的递归数量与 JavaScript 调用栈大小直接相关。只有 IE 例外，它的调用栈与系统空闲内存有关，而其他所有浏览器都有固定数量的调用栈限制。大多数现代浏览器的调用栈数量比老版本浏览多出很多（比如 Safari 2 的调用栈大小只有 100）。图 4-2 展示了主流浏览器调用栈的大小对比。

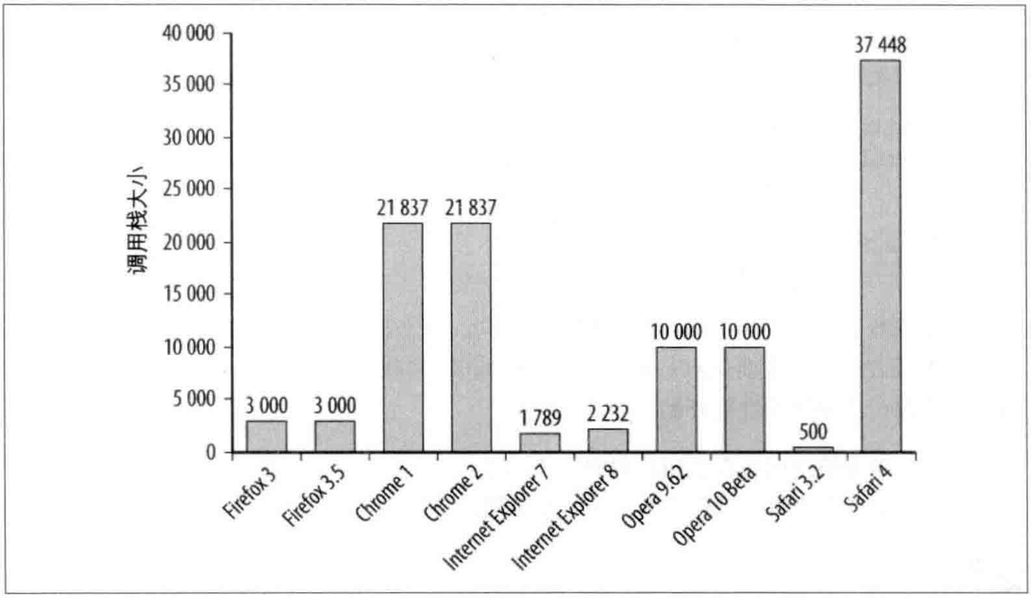


图 4-2　各种浏览器的 JavaScript 调用栈大小对比

当你使用了太多的递归，甚至超过最大调用栈容量时，浏览器会报告以下出错信息：

- Internet Explorer: “Stack overflow at line x”

- Firefox: “Too much recursion”
- Safari: “Maximum call stack size exceeded”
- Opera: “Abort (control stack overflow)”

Chrome 是唯一不显示调用栈溢出错误的浏览器。

关于调用栈溢出错误，也许最有趣的部分是，在某些浏览器中，它们的确是 JavaScript 错误，因此能用 `try-catch` 表达式捕获。异常类型也因浏览器而不同。在 Firefox 中，它是一个内部错误（InternalError）；在 Safari 和 Chrome 中，它是一个 `RangeError`；在 IE 中则抛出一个常规的 `Error` 类型。(Opera 只会中止 JavaScript 引擎，不抛出错误。) 这使得我们能在 JavaScript 中正确处理这些错误：

```
try {
    recurse();
} catch (ex){
    alert("Too much recursion!");
}
```

如果不捕获它，这些错误会像其他错误一样向上冒泡传递（在 Firefox 中，冒泡停止于 Firebug 和错误控制台），在 Safari/Chrome 中错误会显示在 JavaScript 控制台上）。只有 IE 例外，它不但显示一个 JavaScript 错误，还会弹出一个类似 alert 警告的对话框显示栈溢出信息。

提示：尽管在 JavaScript 中捕获这些错误是有可能的，但并不推荐这样做。那些有潜在的调用栈溢出问题的脚本就不应该发布上线。

递归模式

Recursion Patterns

当你遇到调用栈大小限制时，第一步应该先检查代码中的递归实例。为此，有两种递归模式值得注意。第一种是以前面提到的`factorial()`函数为代表的直接递归模式，即函数调用自身。一般写法如下：

```
function recurse(){
    recurse();
}

recurse();
```

在发生错误时，这个模式很容易定位。另一种模式称为“隐伏模式”，它包含两个函数：

```
function first(){
    second();
}

function second(){
    first();
}

first();
```

在这种递归模式中，两个函数相互调用，形成一个无限循环。这种模式更令人不安，在大型代码库中很难定位原因。

大多数调用栈错误都与这两种模式有关。最常见的导致栈溢出的原因是不正确的终止条件，因此定位模式错误的第一步是验证终止条件。如果终止条件没问题，那么可能是算法中包含了太多层递归，为了能在浏览器中安全地工作，建议改用迭代、Memoization [译注4]，或者结合两者使用。

迭代

Iteration

任何递归能实现的算法同样可以用迭代来实现。迭代算法通常包含几个不同的循环，分别对应计算过程的不同方面，这也会引入它们自身的性能问题。然而，使用优化后的循环替代长时间运行的递归函数可以提升性能，因为运行一个循环比反复调用一个函数的开销要少得多。

例如，合并排序算法是最常见的用递归实现的算法。用如下 JavaScript 代码实现一个简单的合并排序：

```
function merge(left, right){
    var result = [];

    while (left.length > 0 && right.length > 0){
        if (left[0] < right[0]){
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }

    return result.concat(left).concat(right);
}

function mergeSort(items){

    if (items.length == 1) {
        return items;
    }

    var middle = Math.floor(items.length / 2),
        left = items.slice(0, middle),
```

[译注4] Memoization 又称为 Tabulation，它是一种主要用于加速程序计算的优化技术，它使得函数避免重复演算之前已被处理过的参数，而返回已缓存的结果。详见维基百科：*http://en.wikipedia.org/wiki/Memoization*。

```
        right = items.slice(middle);

    return merge(mergeSort(left), mergeSort(right));
}
```

这段合并排序的代码相当简单直观，但是 `mergeSort()`函数会导致很频繁的自调用。一个长度为 n 的数组最终会调用 `mergeSort()` 2^*n-1 次，这意味着一个长度超过 1500 的数组会在 Firefox 上发生栈溢出错误。

程序遇到栈溢出错误并不一定要修改整个算法，这只是表明递归并不是最好的实现方式。这个合并排序算法同样可以用迭代实现，比如：

```
//使用与前例相同的 mergeSort()函数

function mergeSort(items){

    if (items.length == 1) {
        return items;
    }

    var work = [];
    for (var i=0, len=items.length; i < len; i++){
        work.push([items[i]]);
    }
    work.push([]);  //如果数组长度为奇数

    for (var lim=len; lim > 1; lim = (lim+1)/2){
        for (var j=0,k=0; k < lim; j++, k+=2){
            work[j] = merge(work[k], work[k+1]);
        }
        work[j] = [];  // 如果数组长度为奇数
    }

    return work[0];
}
```

这个版本的 `mergeSort()`函数功能与前例相同却没有使用递归。尽管迭代版本的合并排序算法比递归实现得要慢一些，但它不会像递归版本那样受调用栈限制的影响。把递归算法改用迭代实现是避免栈溢出错误的方法之一。

Memoization

减少工作量就是最好的性能优化技术。代码要处理的事越少，它的运行速度就越快。沿着这个思路，避免重复工作也是有意义的。多次执行相同的任务纯粹是浪费时间。Memoization 正是一种避免重复工作的方法，它缓存前一个计算结果供后续计算使用，避免了重复工作。这使得它成为递归算法中有用的技术。

当代码运行过程中多次调用递归函数时，大量重复工作不可避免。`factorial()`函数（前面第73页“递归”章节有过介绍）是一个递归函数重复工作的典型示例。考虑如下代码：

```
var fact6 = factorial(6);
var fact5 = factorial(5);
var fact4 = factorial(4);
```

这段代码有3次阶乘计算，导致`factorial()`函数一共被调用了18次。该代码中最糟糕的部分是，所有必要的计算在第一行代码里就已经处理掉了。由于6的阶乘等于6乘以5的阶乘，因此5的阶乘被计算了两次。更糟糕的是，4的阶乘被计算了3次。更为明智的做法是保存并重用它们的计算结果，而不是每次都重新计算整个函数。

你可以利用Memoization技术来重写`factorial()`函数，代码如下：

```
function memfactorial(n){

    if (!memfactorial.cache){
        memfactorial.cache = {
            "0": 1,
            "1": 1
        };
    }

    if (!memfactorial.cache.hasOwnProperty(n)){
        memfactorial.cache[n] = n * memfactorial (n-1);
    }

    return memfactorial.cache[n];
}
```

这个优化版本的factorial function的关键在于创建一个缓存对象。这个对象存储在函数自身内部，并预设两个最简单的阶乘：0和1。在计算一个阶乘之前，首先检查这个缓存对象看看是否已经存在相应的计算结果。没有对应的缓存值则意味着这是第一次计算，计算完成之后，结果被储存在缓存中为以后所用。这个函数的用法同原始的`factorial()`函数相同：

```
var fact6 = memfactorial(6);
var fact5 = memfactorial(5);
var fact4 = memfactorial(4);
```

该代码返回了3个不同的阶乘值，但只调用了`memfactorial()`函数8次。因为所有必要的计算都在第一行完成并缓存了，所以接下来的两行代码不会发生递归运算，而是直接返回缓存中的值。

Memoization 的过程和递归函数稍有不同，但原理是相同的。要想 memoizing 函数变得更简单，你可以定义一个封装了基础功能的 `memoize()` 函数。例如：

```
function memoize(fundamental, cache){
    cache = cache || {};

    var shell = function(arg){
        if (!cache.hasOwnProperty(arg)){
            cache[arg] = fundamental(arg);
        }
        return cache[arg];
    };

    return shell;
}
```

`memoize()` 函数接收两个参数：一个是需要增加缓存功能的函数，一个是可选的缓存对象。如果你需要预设一些值，就给缓存对象传入一个预设的缓存对象，否则会创建一个新的缓存对象。然后创建一个封装了原始函数（`fundamental`）的外壳（shell）函数，以确保只有当一个结果值之前从未被计算过时才会产生新的计算。最后这个外壳函数被返回，你可以直接调用它，比如：

```
// 缓存该阶乘函数
var memfactorial = memoize(factorial, { "0": 1, "1": 1 });

// 调用新函数
var fact6 = memfactorial(6);
var fact5 = memfactorial(5);
var fact4 = memfactorial(4);
```

这种通用的 Memoization 与手工更新给定函数的算法相比优化效果要差一些，因为 `memoize()` 函数会缓存特定参数的函数调用结果。当代码以相同的参数多次调用外壳函数时才能节省时间。因此，当 Memoization 函数存在显著性能问题时，最好有针对性地手工实现它，而不是直接用通用 Memoization 方案。

小结

Summary

JavaScript 和其他编程语言一样，代码的写法和算法会影响运行时间。与其他语言不同的是，JavaScript 可用资源有限[译注5]，因此优化技术更为重要。

- `for`、`while` 和 `do-while` 循环性能特性相当，并没有一种循环类型明显快于或慢于其他类型。
- 避免使用 `for-in` 循环，除非你需要遍历一个属性数量未知的对象。

译注5：由于 JavaScript 是解释性语言，与编译性语言不同的是，它无须编译，而是将代码以字符串的形式交给 JavaScript 引擎来执行。因此，代码性能在一定程度上取决于客户端浏览器的 JavaScript 引擎。

- 改善循环性能的最佳方式是减少每次迭代的运算量和减少循环迭代次数。
- 通常来说，`switch` 总是比 `if-else` 快，但并不总是最佳解决方案。
- 在判断条件较多时，使用查找表比 `if-else` 和 `switch` 更快。
- 浏览器的调用栈大小限制了递归算法在 JavaScript 中的应用；栈溢出错误会导致其他代码中断运行。
- 如果你遇到栈溢出错误，可将方法改为迭代算法，或使用 Memoization 来避免重复计算。

运行的代码数量越大，使用这些策略所带来的性能提升也就越明显。

第 5 章

字符串和正则表达式

Strings and Regular Expressions

Steven Levithan

几乎所有的 JavaScript 程序都与字符串操作密切相关。例如，许多应用使用 Ajax 从服务端获取字符串，并把这些字符串转换为更易用的 JavaScript 对象，然后由这些数据生成 HTML 字符串。一个典型的应用程序通常需要处理大量类似合并、分割、重新排序、搜索、遍历等字符串操作。随着 Web 应用越来越复杂，越来越多的类似任务会在浏览器中完成。

在 JavaScript 中，正则表达式是必不可少的，它的重要性远远超过那些琐碎的字符串处理。本章大部分内容会帮助你更好地理解正则表达式引擎[注1]的内部字符串处理机制，并告诉你如何利用这些知识编写高效的正则表达式。

提示：由于名词“regular expression”有些长，通常简写成“regex”。

此外，在本章你还会学到最高效且跨浏览器的合并与去除字符串首尾空白的方法，探索如何通过减少回溯次数来提升正则表达式性能，并了解大量关于高效处理字符串和正则表达式的各种技巧。

字符串连接

String Concatenation

字符串连接会导致令人惊讶的性能问题。构建字符串的常用方法是：通过一个循环，

注1：　该引擎就是个能运行正则表达式的软件。每个浏览器都实现了它自己的正则表达式引擎（当然，如果你愿意的话也可以自己来实现），它们都有着各自独特的优势。

向字符串末尾不断地添加内容（例如，构建一个 HTML 表格或 XML 文档）。但这类方法因在某些浏览器中性能糟糕而臭名昭著。

那么，怎样优化这类任务？对于初学者，有多种方法可以合并字符串（详见表 5-1）。

表 5-1　字符串合并的方法

方法	示例
The + operator	`str = "a" + "b" + "c";`

The += operator	`str = "a";`
	`str += "b";`
	`str += "c";`

`array.join()`	`str = ["a", "b", "c"].join("");`
`string.concat()`	`str = "a";` `str = str.concat("b", "c");`

当连接少量字符串时，这些方法运行速度都很快，使用时可以选择最熟悉的做法。随着需要合并的字符串的长度和数量增加，有一些方法开始展现出优势。

加（+）和加等（+=）操作符

Plus (+) and Plus-Equals (+=) Operators

这些操作符提供了连接字符串最简单的方法，事实上，除 IE 7 及早期版本外的所有现代浏览器，都对它们进行了良好的优化，所以你真的没必要寻找其他方法。然而，有些技术能使这些操作的效率最大化。

首先，看一个例子。这是一个连接字符串的常用方法：

```
str += "one" + "two";
```

此代码运行时，会经历四个步骤：

1. 在内存中创建一个临时字符串。
2. 连接后的字符串“`onetwo`”被赋值给该临时字符串。
3. 临时字符串与 `str` 当前的值连接。
4. 结果赋值给 `str`。

这是浏览器完成此任务的大概步骤。

以下代码用两行语句直接附加内容给 `str`，从而避免了产生临时字符串（列表中的第一步和第二步）。在大多数浏览器中这样做会提速 10%～40%：

```
str += "one";
str += "two";
```

事实上，你可以只用一个语句就能达到同样的性能提升，看如下代码：

```
str = str + "one" + "two";
// 等价于 str = ((str + "one") + "two")
```

赋值表达式由 str 开始作为基础，每次给它附加一个字符串，由左向右依次连接，因此避免了使用临时字符串。如果改变连接顺序（例如：`str = "one" + str + "two"`），本优化将失效。这与浏览器合并字符串时分配内存的方法有关。除 IE 外，其他浏览器会尝试为表达式左侧的字符串分配更多的内存，然后简单地将第二个字符串拷贝至它的末尾（见图 5-1）。如果在一个循环中，基础字符串位于最左端的位置，就可以避免重复拷贝一个逐渐变大的基础字符串[译注1]。

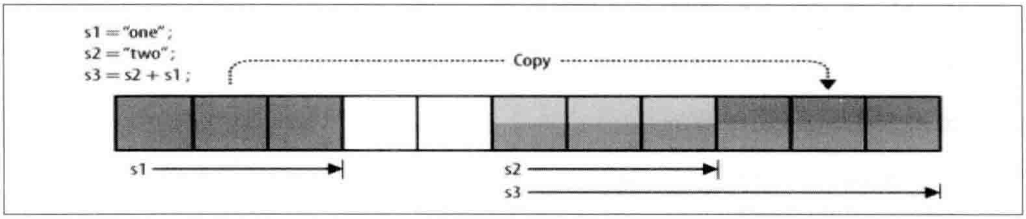


图 5-1　连接字符串时的内存使用示例：s1 被拷贝到 s2 的尾部以创建 s3，基础字符串 s2 并没有被复制

这些技术并不适用于 IE。它们对 IE 8 来说有可能有一点效果，但在 IE 7 及早期版本中结果只会更慢。这是由 IE 执行连接操作的底层机制决定的。在 IE 8 的实现中，连接字符串只是记录现有的字符串的引用来构造新字符串。在最后时刻（当你真正要使用连接后的字符串时），字符串的各个部分才会逐个拷贝到一个新的“真正的”字符串中，然后用它取代先前的字符串引用，所以并非每次使用字符串时都发生合并操作。

提示：IE 8 的实现可以骗过复合基准测试——让连接过程看上去比实际更快——除非你在每次完成测试字符串构建后强制进行连接。比如，调用最终字符串的 `toString()` 方法，或检查它的 `length` 属性，或把它插入 DOM 里。

IE 7 及更早期版本在连接字符串时使用了更糟糕的实现方法：每连接一对字符串都要把它复制到一块新分配的内存中。你会在后面的“数组合并”章节看到它潜在的巨大影响。

[译注1] 基本字符串可理解为连接时排在前面的字符串。文中的意思是：str + one 意味着拷贝 one 并附加在 str 之后，而 one + str 则意味着要拷贝 str 并附加在 one 之后，str 如果很大，拷贝过程的性能损耗（内存占用）就会很高。

针对 IE 8 之前的浏览器，本章提供的建议反而会使代码更慢，因为在与一个长的基础字符串合并前先连接多个短字符串会使速度更快（避免了多次复制大字符串）。例如，`largeStr = largeStr + s1 + s2`，IE 7 及早期版本中，必须将这个长字符串拷贝两次，首先是与 `s1` 合并，然后再与 `s2` 合并。相反，`largeStr += s1 + s2` 首先将两个小字符串合并起来，然后将结果返回给长字符串。创建中间字符串 `s1+s2` 与两次拷贝字符串相比，性能影响要小得多。

Firefox 和编译期合并

在赋值表达式中所有要连接的字符串都属于编译期常量，Firefox 会在编译过程中自动合并他们。有个方法可以看到这一过程：

```
function foldingDemo() {
    var str = "compile" + "time" + "folding";
    str += "this" + "works" + "too";
    str = str + "but" + "not" + "this";
}

alert(foldingDemo.toString());

/* 在 Firefox 中，将会看到如下情况：
function foldingDemo() {
    var str = "compiletimefolding";
    str += "thisworkstoo";
    str = str + "but" + "not" + "this";
} */
```

当字符串通过这种方式合并在一起时，由于运行期没有中间字符串，所以花在连接过程的时间和内存可以减少到零。这种做法非常不错，但它不是经常起作用，因为更多的时候是用运行期的数据构建字符串，而不是用编译期常量。

提示：YUI Compressor 在重构代码时会执行这个优化。详见“JavaScript 压缩”章节的 168 页内容。

数组项合并

Array Joining

`Array.prototype.join` 方法将数组的所有元素合并成一个字符串，它接收一个字符串参数作为分隔符插入每个元素的中间。如果传入的分隔符为空字符，你可以简单地将数组所有元素连接起来。

在大多数浏览器中，数组项合并比其他字符串连接方法更慢，但事实上，它却是在 IE 7 及更早版本浏览器中合并大量字符串唯一高效的途径，这也算是一种补偿。

下列示例代码演示了数组项合并解决的性能问题：

```
var str = "I'm a thirty-five character string.",
    newStr = "",
    appends = 5000;

while (appends--) {
    newStr += str;
}
```

此代码连接了 5 000 个长度为 35 的字符串。图 5-2[注2] 显示了在 IE 7 中完成测试所消耗的时间，从 5 000 次连接开始，逐步增加连接数量。

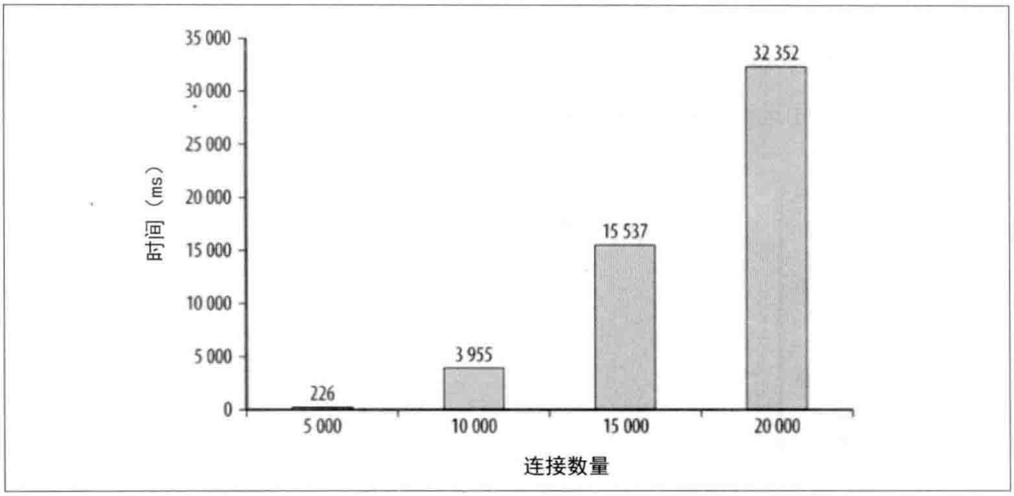


图 5-2 在 IE 7 中使用+=连接字符串所用的时间

IE 7"天真"的连接算法要求浏览器在循环过程中为逐渐增大的字符串不断复制并分配内存。结果是运行时间和内存消耗以平方关系递增。

好消息是在其他所有的主流浏览器（包括 IE 8）在本测试中表现优异，没有呈现出平方关系的复杂性递增，这是个真正的杀手级优化。然而，此代码演示了看似简单的字符串连接所产生的影响。5 000 次合并用了 226 毫秒已经明显地影响到性能，应该尽可能缩短这一时

注 2：图 5-2 和图 5-3 的数字是由具有主流配置（2GHz Core 2 Duo CPI 和 1 GB 内存）的 Windows XP 虚拟机中的 IE 7 每运行测试 10 次的平均数生成。

间，在任何程序中只是为了连接 20 000 个短字符串而导致浏览器假死 32 秒以上都是完全难以接收的。

现在考虑下面的测试用例，它使用数组项合并生成相同的字符串：

```
var str = "I'm a thirty-five character string.",
    strs = [],
    newStr,
    appends = 5000;

while (appends--) {
    strs[strs.length] = str;
}

newStr = strs.join("");
```

图 5-3 显示在 IE 7 中运行此测试所用的时间。

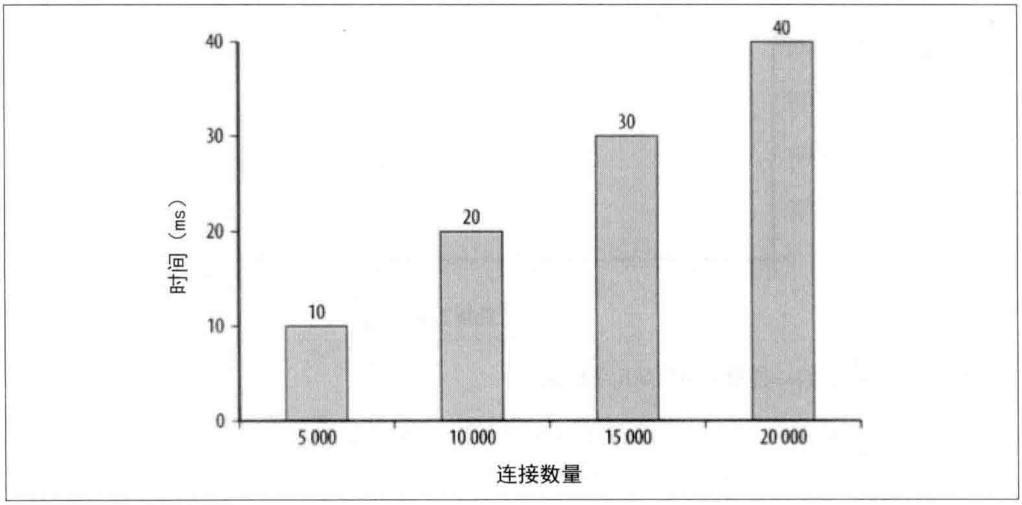


图 5-3 在 IE 7 中使用数组合并的方法所用的时间

由于避免了重复分配内存和拷贝逐渐增大的字符串，于是有了这个戏剧性的性能提升。当把所有的数组的元素连接在一起时，浏览器会分配足够的内存来存放整个字符串，而且不会多次拷贝最终字符串中相同的部分。

String.prototype.concat

字符串的原生方法 `concat` 能接收任意数量的参数，并将每一个参数附加到所调用的字符串上。这是最灵活的字符串合并方法，因为你可以用它只附加一个字符串，或者同时附加多个字符串，以至整个字符串数组。

```
// 附加一个字符串
str = str.concat(s1);

// 附加三个字符串
str = str.concat(s1, s2, s3);

// 如果传递一个数组，可以附加数组中的所有字符串
str = String.prototype.concat.apply(str, array);
```

遗憾的是，在多数情况下，使用 `concat` 比使用简单的 `+` 和 `+=` 稍慢，尤其是在 IE、Opera 和 Chrome 中慢得更明显。此外，尽管使用 `concat` 合并字符串数组与前面讨论的数组项连接类似，但它通常更慢一些（Opera 除外），并且它也潜伏着灾难性的性能问题，就像在 IE 7 及早期版本中使用 `+` 和 `+=` 构建大字符串时那样。

正则表达式优化

Regular Expression Optimization

草率地编写正则表达式可能是造成性能瓶颈的主要原因（后面的“回溯失控”一节有些例子说明这个问题的严重性），但也有很多提高正则表达式运行效率的方法。两个正则表达式匹配相同的文本并不意味着它们有着同样的速度。

正则表达式的运行效率受许多因素影响。首先，正则表达式匹配的文本千差万别，部分匹配比完全不匹配所用的时间要长。不同的浏览器对正则表达式引擎有着不同程度的内部优化[译注 2]。

正则表达式优化是个相当广泛又细致入微的话题。本节涵盖的范围有限，但希望这些内容能有助于你理解影响正则表达式性能的各种因素，并掌握编写高效正则表达式的艺术。

请注意，本节假定你在使用正则表达式方面已有一些经验，并主要关注于如何使它们运行得更快。如果你是初次接触正则表达式，或许你还需要复习一下基础，有许多书籍和网络资源可供参考。由 Jan Goyvaerts 和 Steven Levithan（本书作者）合著的《正则表达式手册》（“Regular Expressions Cookbook”，*http://oreilly.com/catalog/97805965206941*）（O’Reilly）涵盖了 JavaScript 和其他编程语言，这本书是为敢于实践的人们编写的。

译注 2：　这样的结果是，看似无关紧要的改变导致正则在一个浏览器中更快，而在另一个中却更慢。

正则表达式工作原理

How Regular Expressions Work

为了更高效地使用正则表达式，首先要理解它的工作原理。下面是一个正则表达式处理的基本步骤。

第一步：编译

当你创建了一个正则表达式对象（使用正则直接量或 `RegExp` 构造函数），浏览器会验证你的表达式，然后把它转化为一个原生代码程序，用于执行匹配工作。如果你把正则对象赋值给一个变量，可以避免重复执行这一步骤。

第二步：设置起始位置

当正则类进入使用状态，首先要确定目标字符串的起始搜索位置。它是字符串的起始字符，或者由正则表达式的 `lastIndex` 属性[译注 3]指定，但是当它从第四步返回到这里时（由于尝试匹配失败），此位置则在最后一次匹配的起始位置的下一位字符的位置上。

浏览器厂商优化正则表达式引擎的办法是，通过提前决定跳过一些不必要的的步骤，来避免大量无意义的工作。举个例子，如果正则表达式由 `^` 开始，IE 和 Chrome 通常会判断字符串的起始位置能否匹配，如果匹配失败，那么可以避免愚蠢地搜索后续位置。另一个例子是匹配第三个字母是 `x` 的字符串，一个聪明的做法是先找到 `x`，然后再将起始位置回退两个字符（例如，最新版本的 Chrome 包含了这项优化）。

第三步：匹配每个正则表达式字元

一旦正则表达式知道开始位置，它会逐个检查文本和正则表达式模式。当一个特定的字元匹配失败时，正则表达式会试着回溯到之前尝试匹配的位置上，然后尝试其他可能的路径。

第四步：匹配成功或失败

如果在字符串当前的位置发现了一个完全匹配，那么正则表达式宣布匹配成功。如果正则表达式所有的可能路径都没有匹配到，正则表达式引擎会回退到第二步，然后从下一个字符重新尝试。当字符串的每个字符（以及最后一个字符串后面的位置）都经历这个过程，如果还没有成功匹配，那么正则表达式就宣布彻底匹配失败。

[译注 3] 正则表达式的 `lastIndex` 属性值只作为 `exec` 和 `test` 方法的起始搜索位置，并且仅当正则表达式带有`/g`（global）标识的时候。非全局或任何正则表达式作为参数传递给字符串的 `match`, `replace`, `search` 及 `split` 方法时，会搜索目标字符串的起始位置。

牢记这个过程有助于你明智地判别各种影响正则表达式性能的问题。接下来我们深入讨论在第三步中提到的匹配过程中的关键特性：回溯。

理解回溯[译注4]

Understanding Backtracking

在大多数现代正则表达式的实现中（包括 JavaScript 所依赖的部分），回溯是匹配过程的基础组成部分。这在很大程度上也是正则表达式如此强大且富有表现力的根源。然而，回溯会产生昂贵的计算消耗，一不小心就会失控。尽管回溯只是影响整体性能的其中一环，但理解它的工作原理以及如何最少化地使用它可能是编写高效正则表达式的关键所在。接下来几节会用较长篇幅讨论这个话题。

当正则表达式匹配目标字符串时，它从左到右逐个测试表达式的组成部分，看是否能找到匹配项。在遇到量词和分支时，[译注5]需要决策下一步如何处理。如果遇到量词（诸如 `*`，`+?` 或 `{2,}`），正则表达式需决定何时尝试匹配更多字符；如果遇到分支（来自 `|` 操作符），那么必须从可选项中选择一个尝试匹配。

每当正则表达式做类似的决定时，如果有必要的话，都会记录其他选择，以备返回时使用。如果当前选项匹配成功，正则表达式继续扫描表达式，如果其他部分也匹配成功，那么匹配结束。但是如果当前选项找不到匹配值，或后面的部分匹配失败，那么正则表达式会回溯到最后一个决策点，然后在剩余的选项中选择一个。这个过程会一直进行，直到找到匹配项，或者正则表达式中量词和分支选项的所有排列组合都尝试失败，那么它将放弃匹配，转而移动到字符串中的下一个字符，再重复此过程。

分支与回溯

下面的例子演示了处理分支的过程：

```
/h(ello|appy) hippo/.test("hello there, happy hippo");
```

这个正则表达式匹配“hello hippo”或“happy hippo”。匹配过程开始时，首先会查找一个 `h`，目标字符串的首字母恰好是 h，于是立刻被找到。接下来，子表达式（`ello|appy`）提供了两个处理选项。正则表达式选择最左侧的选项（分支选择总是从左到右进行），检查 `ello` 是否匹配字符串中的下一个字符，匹配成功，正则表达式进而匹配随后的空格，由于 `hippo`

[译注4] 参见维基百科 *http://zh.wikipedia.org/zh/回溯法*。

[译注5] 虽然类似`[a-z]`的字符集和类似\s 或“点”的速记字符集在处理过程允许差异，但由于它们并没有使用回溯（backtracking）来实现，因此不会遇到相同的性能问题。

中的 h 无法匹配下一个字符串中的 t，因此匹配无法继续。此时，正则表达式还不能放弃，因为它还未尝试完所有的可选项，随后它会回溯到最近的决策点（匹配完首字符 h 后面的位置），并尝试匹配的第二个分支。匹配并没有成功，也没有更多的可选项，所以正则表达式认为从字符串的第一个字符开始匹配是不能成功的，因此从第二个字符开始重新尝试。它没有找到 h，于是继续搜索直到在第 14 个字符的位置匹配到“happy”中的 h，然后会再次进入分支过程，这次未能匹配 ello，但是在回溯并尝试第二个分支过程后，匹配到了整个字符串“happy hippo”（参见图 5-4）。匹配成功。

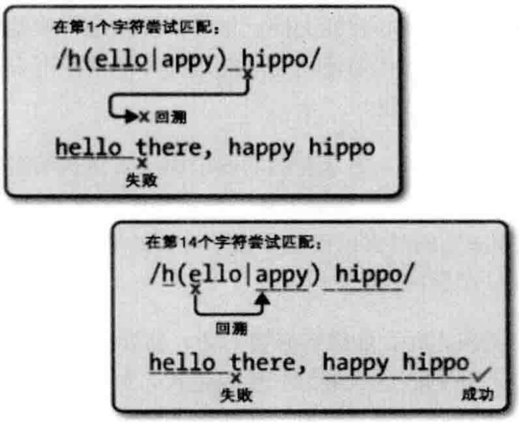

在第1个字符尝试匹配：
/h(ello|appy) hippo/
回溯
hello there, happy hippo
失败
在第14个字符尝试匹配：
/h(ello|appy) hippo/
回溯
hello there, happy hippo
失败
成功

图 5-4 分支回溯的例子

重复与回溯

下面的例子显示了带有重复量词时的回溯过程。

```
var str = "<p>Para 1.</p>" +
          "<img src='smiley.jpg'>" +
          "<p>Para 2.</p>" +
          "<div>Div.</div>";

/<p>.*<\/p>/i.test(str);
```

这个正则表达式一开始就匹配了字符串的前三个字符 `<p>`，然后匹配 `.*` 。点号（.）能匹配除换行符以外的任意字符，贪婪量词星号（*）表示重复零次或多次——尽可能匹配多次。因为目标字符串中没有换行符，那么它会吞并其他所有字符！不过这个正则表达式中仍有更多内容需要匹配，因此正则表达式尝试匹配 <。它在字符串末尾处匹配失败，于是正则

表达式每次回溯一个字符，继续尝试匹配 `<`，直到回溯至 `</div>` 标签的首字符 `<`。接下来尝试匹配 `\/`（转义反斜杠），匹配成功，然后是 `p`，匹配失败。正则表达式继续回溯，并重复这一过程，直到最后匹配到第二段末尾的 `</p>`。匹配成功返回，扫描的范围从第一段头部开始直到最后一段的尾部，这可能并不是你想要的结果。

你可以通过把贪婪量词星号（`*`）替换成惰性（非贪婪）量词 `*?`，以匹配单个段落。惰性量词的回溯工作以相反的方式进行。当正则表达式 `/<p>.*?<\/p>/` 匹配到 `.*?` 时，它先尝试全部跳过并匹配接下来的 `<\/p>`。这样做是因为 `*?` 会重复匹配前一个字符零次或多次，次数尽可能地少，而最小的重复次数就是零次。但是，当紧接着的 `<` 无法匹配当前字符，正则表达式会回溯并尝试匹配下一个最小值：1。它像这样继续向前回溯，直到第一段末尾，在那里量词后面的 `<\/p>` 能够完全匹配。

如果目标字符串只有一个段落，你会发现贪婪与惰性版本的正则表达式是等价的，但是它们的匹配过程并不相同（见图 5-5）。

回溯失控

Runaway Backtracking

当正则表达式导致你的浏览器假死数秒、数分钟、甚至更长时间，问题很可能是因为回溯失控。为了说明这个问题，考虑如下正则表达式，它用来匹配整个 HTML 文件。该正则表达式被拆成多行是为了适应页面显示。与其他正则表达式风格不同的是，JavaScript 没有选项使得点号（.）匹配换行在内的任意字符，因此本例使用 `[\s\S]` 来匹配任意字符。

```
/<html>[\s\S]*?<head>[\s\S]*?<title>[\s\S]*?<\/title>[\s\S]*?<\/head>
[\s\S]*?<body>[\s\S]*?<\/body>[\s\S]*?<\/html>/
```

上述正则表达式在匹配常规的 HTML 字符时运行正常，但是当目标字符串缺少一个或多个必要的标签时会变得很糟糕。比如 `</html>` 标签缺失，最后一个 `[\s\S]*?` 将扩展到字符串末尾，因为仍然没有找到 `</html>` 标签，正则表达式不会放弃，而是依次向前搜索 `[\s\S]*?` 并记住回溯的位置以便后续使用。正则表达式尝试扩展到倒数第二个 `[\s\S]*?` ——用它匹配由正则表达式的 `<\/body>` 模式匹配到的那个 `</body>` 标签——然后继续查找第二个 `</body>` 标签，直到字符串末尾。当所有步骤都失败时，倒数第三个 `[\s\S]*?` 将被扩展至字符串末尾，以此类推。

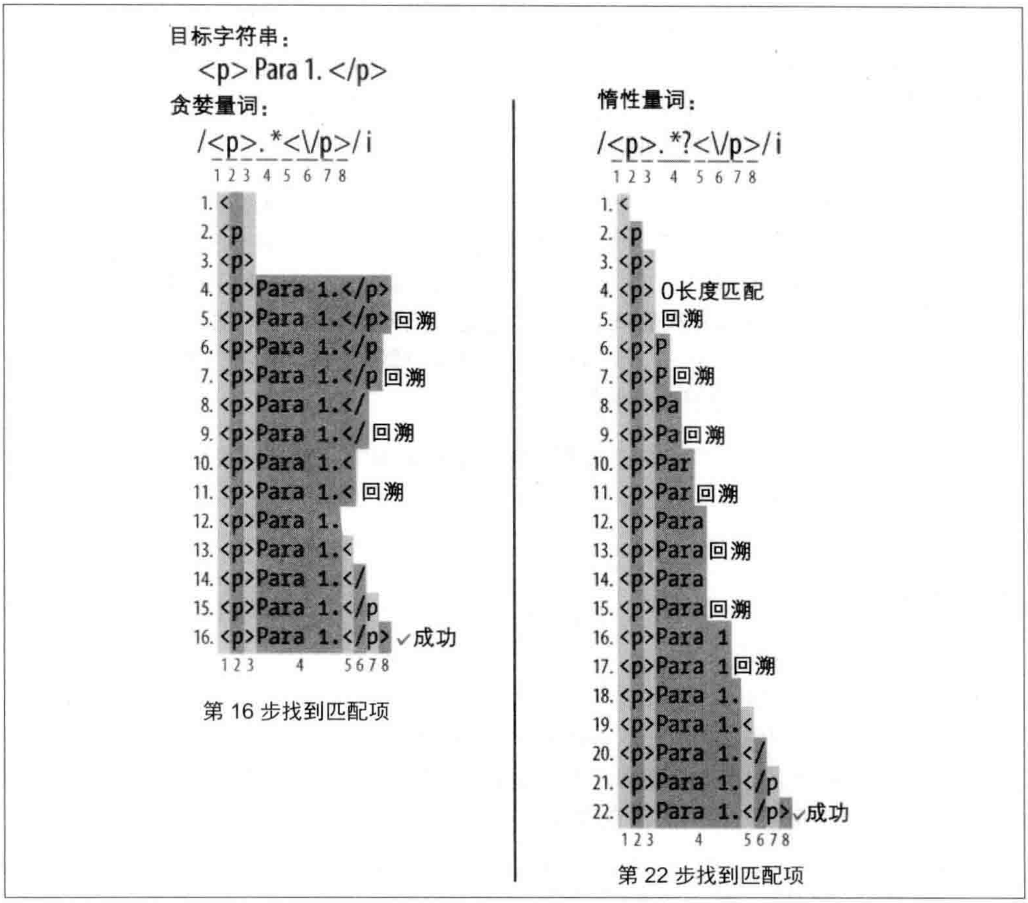

目标字符串：
<p> Para 1. </p>
贪婪量词：
/<p>.*<\/p>/i
1 2 3 4 5 6 7 8
1. <
2. <p
3. <p>
4. <p>Para 1.</p>
5. <p>Para 1.</p> 回溯
6. <p>Para 1.</p
7. <p>Para 1.</p 回溯
8. <p>Para 1.</
9. <p>Para 1.</ 回溯
10. <p>Para 1.<
11. <p>Para 1.< 回溯
12. <p>Para 1.
13. <p>Para 1.<
14. <p>Para 1.</
15. <p>Para 1.</p
16. <p>Para 1.</p> ✓成功
1 2 3 4 5 6 7 8
第 16 步找到匹配项
惰性量词：
/<p>.*?<\/p>/i
1 2 3 4 5 6 7 8
1. <
2. <p
3. <p>
4. <p> 0长度匹配
5. <p> 回溯
6. <p>P
7. <p>P 回溯
8. <p>Pa
9. <p>Pa 回溯
10. <p>Par
11. <p>Par 回溯
12. <p>Para
13. <p>Para 回溯
14. <p>Para
15. <p>Para 回溯
16. <p>Para 1
17. <p>Para 1 回溯
18. <p>Para 1.
19. <p>Para 1.<
20. <p>Para 1.</
21. <p>Para 1.</p
22. <p>Para 1.</p> ✓成功
1 2 3 4 5 6 7 8
第 22 步找到匹配项

图 5-5　贪婪量词和惰性量词回溯的例子

解决方案：具体化

类似问题的解决方案是，尽可能具体化分隔符之间的字符串匹配模式。比如模式“.*?”，它用来匹配一个由双引号包围的字符串。通过把这个过于宽泛的 .*? 替换成更为具体的 [^"\r\n]*，就去除了回溯时可能发生的几种情况，如尝试用点号匹配引号或者扩展搜索超出预期范围。

在 HTML 的示例中，解决办法不是那么简单。你不能使用一个取反字符集如 [^<] 来替代 [\s\S]，因为搜索过程中可能存在其他标签。但是，你可以通过重复一个非捕获组来达到同样的效果，它包含了否定性预查（negative lookahead）（阻止下一个依赖标签）和[\s\S]（任意字符）元序列。这确保了中间位置上你查找的每一个标签都会失败，更重要的是，

表达式 [\s\S] 在你预查过程中阻塞的标签被发现之前不能被扩展。使用此方法后正则表达式最终修改如下：

```
/<html>(?:(?!<head>)[\s\S])*<head>(?:(?!<title>)[\s\S])*<title>
(?:(?!<\/title>)[\s\S])*<\/title>(?:(?!<\/head>)[\s\S])*<\/head>
(?:(?!<body>)[\s\S])*<body>(?:(?!<\/body>)[\s\S])*<\/body>
(?:(?!<\/html>)[\s\S])*<\/html>/
```

虽然这样做消除了潜在的回溯失控，并允许正则表达式在匹配不完整的 HTML 字符串失败时所需要时间同字符串长度成线性关系，但是它的效率并没有提高。像这样为每个匹配字符重复预查是严重缺乏效率的，会明显降低成功匹配的速度。这种方法在匹配短字符串时运行良好，但是本例中为匹配 HTML 文件向前查看可能需要测试上千次。有一种效率更高的解决方案，它依赖一点小技巧，如下所述。

使用预查和反向引用的模拟原子组

一些正则表达式引擎，包括 .NET、Java、Oniguruma、PCRE 和 Perl，都支持一种名为“原子组”的特性。原子组的写法是`(?>...)`，省略号表示任意正则表达式的模式，它是一种具有特殊反转性的非捕获组[译注6]。一旦原子组中存在一个正则表达式，该组的任何回溯位置都会被丢弃。这为 HTML 正则表达式的回溯问题提供了一个更好的解决方案：如果你将 [\s\S]*? 序列和它后面的 HTML 标签放在一个原子组中，每当所需要的 HTML 标签被发现一次，这次匹配基本上就被锁定了。如果该正则表达式的后续部分匹配失败，原子组中的量词不会记录回溯点，因此 [\s\S]*? 序列已经匹配的部分不会再被展开。

这样做很不错，但是 JavaScript 不支持原子组，也没有提供其他方法消除不必要的回溯。不过，你可以利用预查过程中一项鲜为人知的行为来模拟原子组：预查也是原子组[注3]。它们的区别是，预查作为全局匹配的一部分，并不消耗任何字符；而只是检查自己包含的正则符号在当前字符串位置是否匹配。你可以通过把预查的表达式封装在捕获组中并给它添加一个反向引用的方法来避免这一问题。下面是代码示例：

```
(?=(pattern to make atomic))\1
```

以上结构适用于所有用到原子组的表达式。请记住，如果你的正则表达式包含了多个捕获组，那么你需要使用适当的反向引用次数。

[译注6] 原子组的目的是使正则引擎回溯结束得更快一点。因此可以有效地阻止海量回溯。原子组的语法是 (?>正则表达式)。位于 (?>) 之间的所有正则表达式都会被认为是一个单一的正则符号。一旦匹配失败，引擎将会回溯到原子组前面的正则表达式部分。

[注3] 依赖这个向前查看的行为是安全的，因为它在所有正则表达式引擎中表现一致，并且都由 ECMAScript 标准作为依据。

HTML 正则表达式应用此方法后的修改如下：

```
/<html>(?=([\s\S]*?<head>))\1(?=([\s\S]*?<title>))\2(?=([\s\S]*?
<\/title>))\3(?=([\s\S]*?<\/head>))\4(?=([\s\S]*?<body>))\5
(?=([\s\S]*?<\/body>))\6[\s\S]*?<\/html>/
```

现在，如果字符串中缺少结尾的 `</html>` 标签，那么最后一个 `[\s\S]*?` 会展开至字符串末尾，于是该正则表达式会立刻匹配失败，因为没有可返回的回溯点。正则表达式每找到一个中间的标签就会退出一个预查，它在预查过程中丢弃所有回溯位置。接下来的反向引用简单地重新匹配预查过程中发现的字符，并将它们作为实际匹配的一部分。

嵌套量词与回溯失控

所谓的嵌套量词需要额外关注且小心使用，以确保不会引发潜在的回溯失控。嵌套量词是指量词出现在一个自身被重复量词修饰的组中（例如：`(x+)*`）。

嵌套量词并不会造成性能危害。然而，如果你不小心的话，它很容易在尝试匹配字符串的过程中，在内部量词和外部量词之间产生一大堆文本拆解的路径。

例如，如果想要匹配 HTML 标签，你会写出如下正则表达式：

```
/<(?:[^>"']|"[^"]*"|'[^']*')*>/
```

这个例子也许过于简单，因为它不能正确辨别出所有有效和无效的字元，但如果只用来处理有效的 HTML 片段，它应该没问题。与更简单的 `/<[^>]*>/` 相比，它的优势在于涵盖了可能会出现在属性值中的 `>` 字符。它使用非捕获组中的第二个和第三个分支，这些分支一次性匹配所有具有单引号和双引号的属性值，并允许除各自的引号类型外的所有字符出现。

到目前为止，还没有回溯失控的风险，即使遇到了嵌套量词星号（`*`）。分组的每次重复过程中，第二和第三分支选项严格匹配一个带引号的字符串，因此潜在的回溯点数量会随着目标字符串的长度而线性增长。

然而，查看非捕获组的第一个分支：`[^>"']`。它每次只能匹配一个字符，看上去似乎有些低效。你可能会认为在字符串末尾加上一个量词 `+` 会更好一些，这样一来正则表达式在每次组重复过程就会在目标字符串中发现匹配的那些位置上匹配多一个字符，你是对的。通过每次匹配多个字符，你会让正则表达式在成功匹配过程中跳过许多不必要的步骤。

没有比这种改变带来的负面效应更显而易见了。如果正则表达式匹配一个起始字符<，但后面没有>，却能匹配成功，回溯失控就会迅速进入高速档，因为内部量词与外部量词的排列组合产生了数量巨大的分支路径（跟在非捕获组之后）用以匹配<后面的文本。正则表达式在最终放弃匹配之前必须尝试所有的排列组合。请当心这样的情形！

变得更糟。关于嵌套量词会导致回溯失控的一个更极端的例子是，在一个只包含大量 `A` 的字符串中应用正则表达式`/(A+A+)+B/`。尽管这个正则表达式写成 `/AA+B/` 会好一些，为了更好地讨论，设想一下两个 `A` 能匹配同一字符串多少种不同的模式。

当应用到由 10 个 `A` 组成的字符串（`"AAAAAAAAAA"`）上，正则表达式首先使用了第一个 `A+` 匹配了 10 个字符。然后正则表达式回溯一个字符，让第二个 `A+` 匹配最后一个字符。然后这个分组试图重复，但是由于没有更多的 `A`，并且该组的 `+` 量词已经符合最少一次的匹配条件，因此正则表达式开始查找 `B`。没有找到，但还不能放弃，因为还有许多路径没有尝试过。如果第一个 `A+` 匹配了 8 个字符，而第二个 `A+` 只匹配了 2 个，会怎样？或者第一个匹配 3 个字符，而第二个匹配 2 个，然后分组重复两次，又会怎样？如果在分组的第一次重复中，第一个 `A+` 匹配 2 个字符，第二个 `A+` 匹配 3 个字符；然后在第二次重复中，第一个匹配 1 个，第二个匹配 4 个，又会怎样？虽然你我都不会笨到认为经过多次回溯后可以找到那个并不存在的 `B`，但正则表达式只会忠实地一次又一次检查所有这些无用的选择。此正则表达式最坏情况的复杂性是惊人的 O(2n)，也就是 2 的 n 次方，n 表示字符串的长度。在 10 个 `A` 构成的字符串中，正则表达式需要经历 1024 次回溯才能确定匹配失败，如果是 20 个 `A`，该数字会剧增到一百万以上。35 个 `A` 足以把 Chrome、IE、Firefox 和 Opera 挂起 10 分钟（如果还没死机的话）用以处理超过 34 000 000 次回溯以排除正则表达式的各种排列组合。最新版本的 Safari 是个例外，它能够检测正则表达式是否陷入循环，并快速中止匹配（Safari 还规定了回溯的次数的上限，超出则中止匹配尝试）。

预防这类问题的关键是确保正则表达式的两个部分不能对字符串的相同部分进行匹配。对正则表达式而言，重写为 `/AA+B/` 可以修复这个问题，但复杂的正则表达式可能难以避免此类问题。增加一个模拟原子组通常作为终极招式，虽然还有其他解决办法，如果可能的话，尽可能保持你的正则表达式简洁易懂。如果这样做，此正则表达式会改成`/((?=(A+A+))\2)+B/`，并彻底消除了回溯问题。

基准测试[译注7]的说明

A Note on Benchmarking

由于正则表达式的性能受所应用的文本影响而产生很大差异，因此没有简单的方法可以测试正则表达式之间的性能差别。为了得到最好的结果，你需要用各种字符串来测试你的正则表达式，包括不同长度的、不匹配的和近似匹配的。

这也是本章的大量讨论回溯的原因之一。如果没有准确的理解回溯，就无法预测和确定回溯相关问题。为了帮助你早日掌握回溯失控，建议总是用包含特殊匹配的长字符串来测试你的正则表达式。为你的正则表达式构建一些近似但不能完全匹配的字符串，并将它们用在你的测试中。

更多提高正则表达式效率的方法

More Ways to Improve Regular Expression Efficiency

下面是一些提升正则表达式效率的技术，其中大部分已经在关于回溯的讨论中有所涉及。

关注如何让匹配更快失败

正则表达式慢的原因通常是匹配失败的过程慢，而不是匹配成功的过程慢。这是因为，如果你使用正则表达式来匹配一个大字符串的一小部分，该正则表达式匹配失败的位置比匹配成功的位置要多得多。如果一个修改将正则表达式的匹配过程变快而失败过程变慢（比如，通过增加回溯的次数去尝试所有的排列组合），这通常会是个失败的修改。

正则表达式以简单、必需的字元开始

最理想的情况是，一个正则表达式的起始标记应当尽可能快速地测试并排除明显不匹配的位置。这样说来好的起始标记通常是一个锚（`^` 或 `$`）、特定字符串（比如：`x` 或 `\u263A`）、字符类（比如：`[a-z]` 或类似 `\d` 的速记符）和单词边界（`\b`）。如果可能的话，避免以分组或选择字元开头，避免类似 `/one|two/` 的顶层分支，因为它强迫正则表达式识别多种起始字元。Firefox 对起始字元中使用的任何量词都很敏感，这样能更好地优化，例如，以 `\s\s*` 替代 `\s+` 或 `\s{1,}`。其他浏览器大多都优化掉这个差异。

使用量词模式，使它们后面的字元互斥

当字符与字元相邻或子表达式能够重叠匹配时，正则表达式尝试拆解文本的路径数量将增加。为避免这种情况，尽量具体化你的匹配模式。当你想表达 “`[^"\s\n]*`” 时不要使用 ”`.*?`”（它依赖回溯）。

减少分支数量，缩小分支范围

分支使用竖线 `|` 可能要求在字符串的每一个位置上测试所有分支选项。你通常可以通

[译注7] 译注 7： *http://en.wikipedia.org/wiki/Benchmarking.*

过使用字符集和选项组件来减少对分支的需求，或将分支在正则表达式上的位置推后（允许到达分支前出现一些匹配失败）。下面的表格显示出这些技术的例子。

替换前	替换后
`cat|bat`	`[cb]at`
`red|read`	`rea?d`
`red|raw`	`r(?:ed|aw)`
`(.|\r|\n)`	`[\s\S]`

提示：那些能匹配任意字符的字符集（如`[\s\S]`，`[\d\D]`，`[\w\W]`，或者`[\0-\uFFFF]`）实际上等同于`(?:.|\r|\n|\u2028|\u2029)`。其中包括点号匹配不到的四个字符（回车、换行符、行分隔符、段分隔符）。

字符集比分支更快，因为它使用位向量（或其他快速实现方式）而不是回溯。当分支必不可少时，将常用分支放到最前面，如果这样做不影响正则表达式匹配的话。分支选项从左到右依次尝试，一个选项被匹配到的机会越多，你越希望它尽快被检测到。

注意 Chrome 和 Firefox 自动执行其中的部分优化，因此较少受到手工调整的影响。

使用非捕获组

捕获组消耗时间和内存来记录反向引用，并使它保持最新。如果你不需要一个反向引用，可使用非捕获组来避免这些开销，比如用`(?:...)`来替代`(...)`。当需要全文匹配的反向引用时，人们喜欢把正则表达式包装在一个捕获组中。这不是必要的，因为你可以使用其他方法引用全文匹配，比如，使用`regex.exec()`返回数组的第一项，或在替换字符串中使用 `$&`。

用非捕获组来替换捕获组在 Firefox 中影响较小，但是在其他浏览器中处理长字符串时会有较大影响。

只捕获感兴趣的文本以减少后处理

作为上一条的补充说明，如果你需要引用匹配的一部分，应该采取一切手段捕获那些片段，再使用反向引用来处理。例如，如果你正在编写代码处理一个正则表达式匹配到的引号括起来的字符串内容，应该使用 `/"([^"]*)"/` 并使用反向引用处理，而不是使用 `/"[^"]*"/` 然后手动从结果中剥离引号。当在循环中使用时，减少这类工作可以节省大量时间。

暴露必需的字元

为了帮助正则表达式引擎在优化查询过程时做出明智的决策，可尝试让它更容易地判断哪些字元是必需的。当字元应用在子表达式或分支中，正则表达式引擎很难判断他们是不是必需的，有些引擎甚至不做此方面的尝试。例如，正则表达式 `/^(ab|cd)/` 暴露它的字符串起始锚。IE 和 Chrome 注意到了这一点，并阻止正则表达式尝试查找首字符以外的匹配，因此使查找瞬间完成而不用关心字符串长度。然而，由于等价正则表达式 `/(^ab|^cd)/` 没有暴露它的锚 `^`，IE 无法应用同样的优化，最终无意义地搜索字符串并在每一个位置上匹配。

使用合适的量词

正如前一节“重复和回溯”中所讨论到的那样，贪婪和惰性量词的匹配过程有较大区别，即便是处理相同的字符串。使用更合适的量词类型（基于预期的回溯数量）可以显著提升性能，尤其是在处理长字符串时。

惰性量词在 Opera 9.*x* 及更早版本中非常慢，但是 Opera 10 修复了这个缺陷。

把正则表达式赋值给变量并重用它们

将正则表达式赋给变量以避免对它们重新编译。有的人更是夸张地使用正则表达式缓存策略，以避免对给定的模式和标记组合进行多次编译。其实不必如此，正则表达式编译很快，这样的策略所增加的负担可能超过它们所避免的。重要的是避免在循环体中重复编译正则表达式。换句话说，别像以下这样：

```
while (/regex1/.test(str1)) {
    /regex2/.exec(str2);
    ...
}
```

而要用下面的做法来替代：

```
var regex1 = /regex1/,
    regex2 = /regex2/;
while (regex1.test(str1)) {
    regex2.exec(str2);
    ...
}
```

将复杂的正则表达式拆分为简单的片段（化繁为简）

避免在一个正则表达式中处理太多任务。复杂的搜索问题需要条件逻辑，拆分成两个或多个正则表达式更容易解决，通常也会更高效，每个正则表达式只在最后的匹配结果中执行查找。在一个模式中完成所有任务的怪兽级正则表达式很难维护，而且容易引起回溯相关的问题。

何时不使用正则表达式

When Not to Use Regular Expressions

当你小心使用时，正则表达式速度非常快。然而，当你只是搜索字面字符串时它经常会弄巧成拙，尤其在你事先知道字符串的哪一部分将要被查找时。比如，如果你要检查一个字符串是否以分号结尾，可能会用到如下表达式：

```
endsWithSemicolon = /;$/.test(str);
```

你可能会惊讶地发现，没有哪个主流浏览器能有如此智能，能够意识到这个正则表达式只匹配字符串的末尾。最终它们所做的将是逐个测试整个字符串。每当找到一个分号，正则表达式就移动到下一个标记（$），检查它是否匹配字符串的末尾。如果不匹配，它会继续搜索匹配项，直到搜索完整个字符串。目标字符串越长（或者包含更多的分号），所花的时间越多。

在这种情况下，一个更好的办法是跳过正则表达式所需的所有中间步骤，简单地检查最后一个字符是否为分号：

```
endsWithSemicolon = str.charAt(str.length - 1) == ";";
```

目标字符串很小时，这种方法只比正则表达式快一点点，但更重要的是，字符串的长度不再影响执行测试所需要的时间。

这个例子使用了 `charAt` 方法来读取特定位置上的字符。字符串方法 `slice`、`substr` 以及 `substring` 都可用在特定位置上提取并检查字符串的值。此外，`indexOf` 和 `lastIndexOf` 方法非常适合查找特定字符串的位置，或者判断它们是否存在。所有的字符串方法速度都很快，当你搜索那些并不依赖正则表达式复杂特性的字面字符串时，它们有助于避免正则表达式带来的性能开销。

去除字符串首尾空白

String Trimming

去除字符串首尾的空白是个简单而常见的任务。虽然 ECMAScript 第五版添加了原生的 `trim` 方法（你会在最新的浏览器中看到它们），但到目前为止 JavaScript 中还没有包含它。对当前浏览器而言，自己实现一个 `trim` 方法或依赖一个包含此功能的库仍然是必要的。

去除字符串首尾空白并非常见的性能瓶颈，但它可作为学习正则表达式优化的例子，因为它有着多种实现方法。

使用正则表达式去首尾空白

Trimming with Regular Expressions

正则表达式能让你用非常简单的代码实现 `trim` 方法，这对关心文件尺寸的 JavaScript 类库

而言十分重要。也许最周全的解决方案是使用两个子表达式——一个去除头部的空白，另一个去除尾部的空白。这种处理简单而快速，特别是在处理长字符串时。

```
if (!String.prototype.trim) {
    String.prototype.trim = function() {
        return this.replace(/^\s+/, "").replace(/\s+$/, "");
    }
}

// 测试新方法
// 头部空白中包含乐制表(\t)和换行符(\n)

var  str = " \t\n test string ".trim();
alert(str == "test string"); // 弹出信息为"true"
```

该示例中 `if` 代码块的作用是避免覆盖已经存在的 `trim` 方法，因为原生的方法是经过优化的，其运行速度通常比你实现的任何 JavaScript 函数都要快。随后的代码范例假设这个条件已经判断过了，所以不会每次都写出来。

通过把 `/\s+$/`（第二个正则表达式）替换成 `/\s\s*$/`，你可以轻易地让 Firefox 的性能提升 35%（或多或少取决于目标字符串的长度和内容）注 4。尽管这两个正则表达式功能完全相同，但 Firefox 却为以非量词标记开始的正则表达式提供了额外的优化。在其他浏览器中，区别不是很明显，或者优化方式不同。不管怎样，修改匹配字符串头部的正则表达式为 `/^\s\s*/` 并不能产生可度量的区别，因为起始的锚^会导致非匹配位置快速失效（避免一个微小的性能差异，因为在一个长字符串中可能产生数千次匹配尝试）。

接下来是一些基于正则表达式的 `trim` 实现，它们是你可能会经常遇到的。你可以查看本段后面的表 5-2，它描述了所有 `trim` 实现在不同浏览器上的性能指数。事实上，你可以编写更多正则表达式来帮助你过滤字符，但是在处理长字符串时，它总是比使用两次简单的表达式要慢（至少在各浏览器中缺乏一致性）。

```
// trim 2
String.prototype.trim = function() {
    return this.replace(/^\s+|\s+$/g, "");
}
```

这可能是最常见的解决方案。它使用分支功能合并了两个简单正则表达式，并使用 /g（全局）标记来替换所有的匹配项，而不是只替换第一个（当目标字符串同时包含首尾空白时

注 4：已通过 Firefox 2、Firefox 3、Firefox 3.5 的测试。

会匹配两次）。这并不是一个很糟糕的方案，但它在处理长字符串时仍然比使用两个简单的子表达式要慢，因为两个分支选项在每个字符串匹配时都要被测试一遍。

```
// trim 3
String.prototype.trim = function() {
    return this.replace(/^\s*([\s\S]*?)\s*$/, "$1");
}
```

这个正则表达式的作用是匹配整个字符串，捕获从第一个到最后一个非空白字符（如果有的话）之间的序列。通过把整个字符串替换成反向引用，剩下的就是修剪后的版本。

这个方法的原理很简单，但是捕获组中的惰性量词会导致正则表达式进行大量额外操作（例如：回溯），因此在处理较长的目标字符串时就变得很慢。正则表达式进入捕获组后，`[\s\S]`类的惰性量词`*?`要求尽可能减少重复次数。因此，正则表达式每匹配一个字符，都要停下来尝试匹配剩下的`\s*$`模式。如果失败了，说明字符串当前位置之后还存在非空白字符，正则表达式将匹配一个或多个字符，更新反向引用，然后再次尝试模式的剩余部分。

惰性重复在 Opera 9.× 及更早的版本中相当慢。因此，在 Opera 9.64 中用此方法过滤长字符串只有其他浏览器处理速度的 1/10 到 1/100。Opera 10 已经修复这个长期存在的缺陷，使得该方法的性能与其他浏览器相比已相差无几。

```
// trim 4
String.prototype.trim = function() {
    return this.replace(/^\s*([\s\S]*\S)?\s*$/, "$1");
}
```

这个与上一个正则表达式类似，但是考虑到性能原因，它把惰性量词替换成了一个贪婪量词。以确保捕获组仍然只匹配最后的非空白字符，末尾的 `\S` 是必需的。然而，由于正则表达式必须能够匹配全部由空格组成的字符串，整个捕获组通过向尾部增加一个 `?` 量词而成为可选组。

在这里，`[\s\S]*`中的贪婪量词星号（`*`）表示重复方括号中的任意字符匹配模式直到字符串末尾。然后正则表达式每次回溯一个字符，直到能够匹配到后面的`\S`，或者直到它回溯到该组中的第一个字符（然后它跳过这个组）。

除非这里的尾部空白比其他文字要多，否则这个方法通常都比前一个使用了惰性量词的方法要快。事实上，它在 IE、Safari、Chrome 和 Opera 10 中都很快，甚至超过使用两个子表达式的方案。那是因为浏览器对能匹配任意字符的贪婪重复有着特别优化。正则表达式引擎跳到字符串末尾而不需要执行中间字符（尽管回溯位置必须记录下来），然后适当回溯。遗憾的是，这个方法在 Firefox 和 Opera 9 中相当慢，至少现在，使用两个子表达式仍然是更好的跨浏览器方案。

```
// trim 5
String.prototype.trim = function() {
    return this.replace(/^\s*(\S*(\s+\S+)*)\s*$/, "$1");
}
```

这是个相当普遍的方法，但在所有浏览器中，它都是上述所有方法中最慢的，因此没有足够的理由去使用它。它与之前两个正则表达式很类似，匹配整个字符串，替换并保留其中一部分，但是由于内部的组每次只匹配一个单词，因此正则表达式需要处理大量的离散步骤。在处理短字符串时，这部分的性能影响几乎无法觉察，但在处理包含很多单词的长字符串时，会导致性能问题。

把内部组改变成非捕获组——比如，把 `(\s+\S+)` 改为 `(?:\s+\S+)`——会对性能有所帮助，在 Opera、IE、Chrome 中能减少 20%~45% 的时间，而 Safari 和 Firefox 中也有轻微改善。尽管如此，一个非捕获组不会完全改善这种实现方式的性能。请注意，由于外部组在被替换字符串中被引用，因此它无法转换成一个非捕获组。

不使用正则表达式去除字符串首尾空白

Trimming Without Regular Expressions

尽管正则表达式速度很快，但不使用正则表达式时性能如何也是值得考虑的。下面是一种做法：

```
// trim 6
String.prototype.trim = function() {
    var start = 0,
        end = this.length - 1,
        ws = " \n\r\t\f\x0b\xa0\u1680\u180e\u2000\u2001\u2002\u2003
\u2004\u2005\u2006\u2007\u2008\u2009\u200a\u200b\u2028\u2029\u202f
\u205f\u3000\ufeff";

    while (ws.indexOf(this.charAt(start)) > -1) {
        start++;
    }
    while (end > start && ws.indexOf(this.charAt(end)) > -1) {
        end--;
    }

    return this.slice(start, end + 1);
}
```

代码中的 `ws` 变量包含了 ECMAScript 第五版定义的所有空白字符。出于性能考虑，在得到修剪后的起始和终止的位置之前避免拷贝字符串的任何部分。

当字符串的末尾只有一小段空白时，正则表达式会陷入疯狂工作状态。这是因为尽管正则

表达式很好地去除了字符串头部的空白，但它们却不能同样快速地去除长字符串尾部的空白。正如“何时不应该使用正则表达式”小节所提到的，正则表达式无法直接跳到字符串末尾而不考虑沿途的字符。然而，本例中却实现了，第二个 `while` 循环从字符串末尾向前寻找，直到找到一个非空白字符为止。

尽管这个版本的性能不受字符串的总长度影响，但它有自己的弱点：不宜用来处理前后大段的空白字符。这是因为通过循环遍历字符串来确定空白字符的效率比不上正则表达式使用的优化后的搜索代码。

混合解决方案

A Hybrid Solution

本节最后的一个方法是把两者结合，用正则表达式方法过滤头部空白，用非正则表达式的方法过滤尾部字符。

```
// trim 7
String.prototype.trim = function() {
    var str = this.replace(/^\s+/, ""),
        end = str.length - 1,
        ws = /\s/;

    while (ws.test(str.charAt(end))) {
        end--;
    }

    return str.slice(0, end + 1);
}
```

这种混合方法在过滤一小段空白时速度非常快，在处理头部有很多空白或者仅由空白组成的字符串时，也没有性能风险（尽管在处理尾部长空白时仍然存在不足）。请记住，该方案在循环中使用一个正则表达式来检查字符串的末尾字符是否为空白。尽管在这里使用正则表达式会带来一些性能开销，但它让你直接使用浏览器定义的空白字符列表，以保持简洁和更好的兼容性。

所有的 `trim` 方法的总的趋势是：在基于正则表达式的方案中，字符串的总长度比修剪掉的字符数量更影响性能；而非正则表达式方案从字符串末尾反向查找，不受字符串总长度的影响，但明显受到修剪空格的数量的影响。简单地使用两次子正则表达式在各种浏览器处理不同内容及长度的字符串是，提供了更一致的性能表现，所以它被证明是最周全的解决方案。混合方案在处理长字符串时特别快，其代价是代码稍长，在某些浏览器上处理尾部长空白时存在不足。更多细节请参阅表 5-2。

表 5-2　不同 trim 实现版本在各种浏览器上的性能

浏览器类别	时间（毫秒）[a]						
	Trim 1[b]	Trim 2	Trim 3	Trim 4	Trim 5[c]	Trim 6	Trim 7
IE 7	80/80	315/312	547/539	36/42	218/224	14/1015	18/409
IE 8	70/70	252/256	512/425	26/30	216/222	4/334	12/205
Firefox 3	136/147	164/174	650/600	1098/1525	1416/1488	21/151	20/144
Firefox 3.5	130/147	157/172	500/510	1004/1437	1344/1394	21/332	18/50
Safari 3.2.3	253/253	424/425	351/359	27/29	541/554	2/140	5/80
Safari 4	37/37	33/31	69/68	32/33	510/514	<0.5/29	4/18
Opera 9.64	494/517	731/748	9066/9601	901/955	1953/2016	<0.5/210	20/241
Opera 10	75/75	94/100	360/370	46/46	514/514	2/186	12/198
Chrome 2	78/78	66/68	100/101	59/59	140/142	1/37	24/55

[a] 报告的时间是修剪一个大字符串（40KB）100 次所用的时间，每个字符串以 10 个空格开头，以 1000 个空格结尾。

[b] 测试时关闭`/\s\s*$/`优化。

[c] 测试时关闭非捕获组优化。

小结

Summary

密集的字符串操作和草率地编写正则表达式可能产生严重的性能障碍，本章提供的建议会帮助你避免这些常见的陷阱。

- 当连接数量巨大或尺寸巨大的字符串时，数组项合并是唯一在 IE 7 及更早版本中性能合理的方法。
- 如果不需考虑 IE 7 及更早版本的性能，数组项合并是最慢的字符串连接方法之一。推荐使用简单的+和+=操作符替代，避免不必要的中间字符串。
- 回溯既是正则表达式匹配功能的基本组成部分，也是正则表达式的低效之源。
- 回溯失控发生在正则表达式本应快速匹配的地方，但因为某些特殊的字符串匹配动作导致运行缓慢甚至浏览器崩溃。避免这个问题的办法是：使相邻的字元互斥，避免嵌套量词对同一字符串的相同部分多次匹配，通过重复利用预查的原子组去除不必要的回溯。
- 提高正则表达式效率的各种技术手段会有助于正则表达式更快地匹配，并在非匹配位置上花更少的时间（参见："更多提高正则表达式效率的方法"）。
- 正则表达式并不总是完成工作的最佳工具，尤其当你只搜索字面字符串的时候。

- 尽管有许多方法可以去除字符串的首尾空白，但使用两个简单的正则表达式（一个用来去除头部空白，另一个用来去除尾部空白）来处理大量字符串内容能提供一个简洁而跨浏览器的方法。从字符串末尾开始循环向前搜索第一个非空白字符，或者将此技术同正则表达式结合起来，会提供一个更好的替代方案，它很少受到字符串长度影响。

第 6 章

快速响应的用户界面

Responsive Interfaces

再也没有比点击网页后却毫无动静更令人沮丧的事情了。这个问题源起于那些事务性的 Web 应用，导致我们现在提交表单时几乎总会遇到“请不要重复提交”的消息。用户会本能地去重复尝试那些没有带来明显变化的行为，所以，确保 Web 应用的响应速度是一个非常重要的性能关注点。

第 1 章介绍了浏览器 UI 线程的概念。简而言之，大多数浏览器让一个单线程共用于执行 JavaScript 和更新用户界面。每个时刻只能执行其中一种操作，这意味着当 JavaScript 代码正在执行时用户界面无法响应输入，反之亦然。当 JavaScript 代码执行时，用户界面处于“锁定”状态。管理好 JavaScript 的运行时间对 Web 应用的性能非常重要。

浏览器 UI 线程

The Browser UI Thread

用于执行 JavaScript 和更新用户界面的进程通常被称为“浏览器 UI 线程”（尽管对所有浏览器来说，称为“线程”不一定准确）。UI 线程的工作基于一个简单的队列系统，任务会被保存到队列中直到进程空闲。一旦空闲，队列中的下一个任务就被重新提取出来并运行。这些任务要么是运行 JavaScript 代码，要么是执行 UI 更新，包括重绘和重排（在第三章讨论过）。也许这个进程中最有趣的部分在于每一次输入[译注1]可能会导致一个或多个任务被加入队列。

考虑一个简单的交互，点击按钮，屏幕上显示一条消息：

```
<html>
<head>
    <title>Browser UI Thread Example</title>
</head>
<body>
```

[译注1] 这里的输入包括：响应用户事件，自执行的 JavaScript 代码等。

```
    <button onclick="handleClick()">Click Me</button>
    <script type="text/javascript">

        function handleClick(){
            var div = document.createElement("div");
            div.innerHTML = "Clicked!";
            document.body.appendChild(div);
        }

    </script>
</body>
</html>
```

当示例中的按钮被点击时，它会触发 UI 线程来创建两个任务并添加到队列中。第一个任务是更新按钮的 UI，它需要改变外观以指示它被点击了，第二个任务是执行 JavaScript，包含 `handleClick()`中的代码，唯一被运行的代码就是这个方法和所有被它调用的方法。假设 UI 线程处于空闲状态，第一个任务被提取出来执行更新按钮的外观，然后 JavaScript 任务被提取出来并执行。在运行过程中，`handleClick()` 创建了一个新的 `<div>` 元素并把它附加在 `<body>` 元素末尾，这实际上引发了另一次 UI 变化。这意味着，在 JavaScript 运行过程中，一个新的 UI 更新任务被添加在队列中，当 JavaScript 运行完之后，UI 还会再更新一次。如图 6-1 所示：

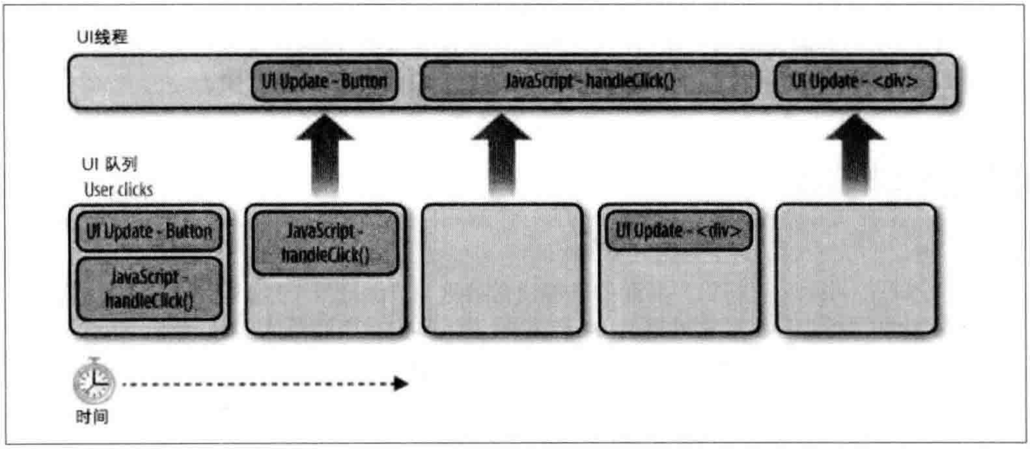

UI线程
UI Update - Button
JavaScript - handleClick()
UI Update - <div>
UI 队列
User clicks
UI Update - Button
JavaScript - handleClick()
JavaScript - handleClick()
UI Update - <div>
时间

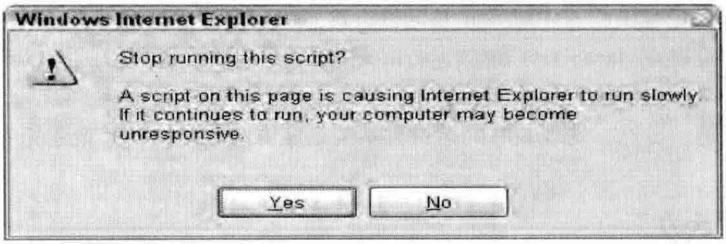
图 6-1 用户与页面交互时向 UI 线程添加的任务

当所有 UI 线程任务都执行完毕，进程进入空闲状态，并等待更多任务加入队列。空闲状态是理想的，因为用户所有的交互都会立刻触发 UI 更新。如果用户试图在任务运行期间与页面交互，不仅没有即时的 UI 更新，甚至可能新的 UI 更新任务都不会被创建并加入队

列。事实上，大多数浏览器在 JavaScript 运行时会停止把新任务加入 UI 线程的队列中，也就是说 JavaScript 任务必须尽快结束，以避免对用户体验造成不良影响。

浏览器限制
Browser Limits

浏览器限制了 JavaScript 任务的运行时间。这种限制是有必要的，它确保某些恶意代码不能通过永不停止的密集操作锁住用户的浏览器或计算机。此类限制分两种：调用栈大小限制（在第 4 章讨论过）和长时间运行（long-running）脚本限制。长时间运行脚本限制有时被称为长时间运行脚本定时器或失控脚本定时器，但其基本原理是浏览器会记录一个脚本的运行时间，并在达到一定限度时终止它。当到达此限制时，浏览器会向用户显示一个对话框，如图 6-2。



Stop running this script?

A script on this page is causing Internet Explorer to run slowly. If it continues to run, your computer may become unresponsive.

图 6-2　当运行超过 500 万条语句时，IE 显示的长时间运行脚本警告对话框

有两种方法可以度量脚本运行了多“长”。第一种是记录自脚本开始以来执行的语句的数量。这种方法意味着脚本在不同的机器上可能会有不同的运行时间，因为可用内存和 CPU 速度会影响单个语句的执行时间。第二种方法是记录脚本执行的总时长。在指定时间内可运行的脚本数量也因用户的机器性能而有所差异，但是到达执行时间后，脚本会停止运行。毫无疑问，不同浏览器检测长时间运行脚本的方法会略有不同：

- IE 自第 4 版开始，设置默认限制为 500 万条语句；此限制存放在 Windows 注册表中，叫作 HKEY_CURRENT_USER\Software\Microsoft\InternetExplorer\ Styles\ MaxScriptStatements。

- Firefox 的默认限制为 10 秒；该限制记录在浏览器配置设置中（通过在地址栏输入 `about:config` 访问），键名为 dom.max_script_run_time。
- Safari 的默认限制为 5 秒；该限制无法更改，但是你可以通过 Develop 菜单选择 Disable Runaway JavaScript Timer 来禁用定时器。
- Chrome 没有单独的长运行脚本限制，替代做法是依赖其通用崩溃检测系统来处理此类问题。
- Opera 没有长运行脚本限制，它会继续执行 JavaScript 代码直到结束，鉴于 Opera 的架构，脚本运行结束时不会导致系统不稳定。

当浏览器的长时间运行脚本限制被触发时，会弹出一个对话框提示用户，而不管页面中其他的错误捕获代码。这是一个主要的可用性问题，因为大多数互联网用户并不精通技术，因此会对出错信息感到迷惑，不知道应该选择停止脚本还是允许它继续运行。

如果你的脚本在任意浏览器中触发此对话框，这意味着脚本花了太多时间来完成任务。它还表明用户浏览器在 JavaScript 继续运行时无法响应用户输入。从开发人员的角度来看，没办法改变一个长时间运行脚本对话框的外观；你检测不到它，因此不能用它来判断任何可能出现的问题。显然，处理长时间运行脚本的最佳方法是从一开始就避免它们。

多久才算“太久”
How Long Is Too Long

浏览器允许脚本持续运行好几秒，但这并不意味着你也允许它这样做。事实上，为了更好的用户体验，你的 JavaScript 代码运行的持续时间应当远远小于浏览器的限制。引用 JavaScript 的创造者 Brendan Eich 的话，“如果[JavaScript]运行了整整几秒钟，那么很可能是你做错了什么……”。

如果对运行 JavaScript 而言，几秒钟算久的话，那么多长时间才算合适呢？事实证明，哪怕是一秒钟，对脚本运行而言也太长了。单个 JavaScript 操作花费的总时间（最大值）不应该超过 100 毫秒。这个数字源自 Robert Miller 于 1968 年的研究[注1]。有趣的是，可用性专家 Jakob Nielsen 在他的著作《可用性工程》（Morgan Kaufmann, 1944）[注2]中也提到，

注 1： Miller, R. B., “Response time in man-computer conversational transactions,” Proc. AFIPS Fall Joint Computer Conference, Vol. 33 (1968), 267–277. 请访问 *http://portal.acm.org/citation.cfm?id=1476589.1476628*。

注 2： 请访问 *www.useit.com/papers/responsetime.html*。

这个数字并不随时间变化而变化，事实上，在 Xerox-PARC（施乐公司）[注3] 1991 年的研究中这个数字被再次重申。

Nielsen 指出如果界面在 100 毫秒内响应用户输入，用户会认为自己在“直接操纵界面中的对象”。超过 100 毫秒意味着用户会感到自己与界面失去联系。由于 JavaScript 运行时无法更新 UI，所以如果 JavaScript 运行时间超过 100 毫秒，用户就会感觉失去了对界面的控制。

更复杂的情况是有些浏览器在 JavaScript 运行时不会把 UI 更新任务加入队列。例如，如果你在某些 JavaScript 代码运行时点击按钮，浏览器可能不会把重绘按钮按下状态的任务或点击按钮启动的新 JavaScript 任务加入队列。最终结果是一个失去响应的 UI，表现为“挂起”或“假死”。

各种浏览器的行为大致相同。当脚本执行时，UI 不随用户交互而更新。执行时间段内用户交互行为所引发的 JavaScript 任务被加入队列中，并在最初的 JavaScript 任务完成后依次执行。而这段时间内由用户交互行为引发的 UI 更新会被自动跳过，因为页面中的动态变化部分会被优先考虑。因此，在一个脚本运行期间点击一个按钮，将无法看到它被按下的样式，尽管它的 `onclick` 事件处理器会被执行。

提示：IE 会控制用户交互行为触发的 JavaScript 任务，因此它会识别连续两次的重复的动作。例如，当有脚本运行时点击一个按钮四次，最终按钮的 `onclick` 事件处理器只被调用两次。

尽管浏览器试图在这类情况下做一些更符合逻辑的事情，但所有的这些行为会导致一个不连续的用户体验。因此最好的方法是，限制所有 JavaScript 任务在 100 毫秒或更短的时间内完成，以避免类似情况出现。这种测量应该以你必须支持的最慢的浏览器为基准（关于度量 JavaScript 性能的工具，请参考第 10 章）。

使用定时器让出时间片段

Yielding with Timers

尽管你尽了最大努力，但难免会有一些复杂的 JavaScript 任务不能在 100 毫秒或更短时间内完成。这个时候，最理想的方法是让出 UI 线程的控制权，使得 UI 可以更新。让出控制权意味着停止执行 JavaScript，使 UI 线程有机会更新，然后再继续执行 JavaScript。于是 JavaScript 定时器走进了我们的视野。

注3： Card, S. K., G.G. Robertson, and J.D. Mackinlay, “The information visualizer: An information workspace,” Proc. ACM CHI’91 Conf. (New Orleans: 28 April–2 May), 181–188. 请访问 *http://portal.acm.org/citation.cfm?id=108874*。

定时器基础

Timer Basics

在 JavaScript 中可以使用 `setTimeout()` 和 `setInterval()` 创建的定时器，它们接收相同的参数：要执行的函数和执行前的等待时间（单位为毫秒）。`setTimeout()` 函数创建了一个只执行一次的定时器，而 `setInterval()` 创建了一个周期性重复运行的定时器。

定时器与 UI 线程的交互方式有助于把运行耗时较长的脚本拆分为较短的片段。调用 `setTimeout()`或 `setInterval()`会告诉 JavaScript 引擎先等待一定时间，然后添加一个 JavaScript 任务到 UI 队列。例如：

```
function  greeting(){
    alert("Hello  world!");
}

setTimeout(greeting, 250);
```

这段代码将在 250 毫秒后，向 UI 队列插入一个执行 `greeting()` 函数的 JavaScript 任务。在这时间点之前，所有其他 UI 更新和 JavaScript 任务都会执行。请记住，第二个参数表示任务何时被添加到 UI 队列，而不是一定会在这段时间后执行；这个任务会等待队列中其他所有任务执行完毕才会执行。考虑如下代码：

```
var  button = document.getElementById("my-button");
button.onclick = function(){

    oneMethod();

    setTimeout(function(){
        document.getElementById("notice").style.color = "red";
    }, 250);
};
```

当例子中的按钮被点击，它会调用一个方法并设置一个定时器。改变 `notice` 元素字体颜色的代码被包含在定时器中，并设置为在 250 毫秒后加入队列。这 250 毫秒从 `setTimeout()`调用时开始计算，而不是在整个函数运行结束后才开始。因此，如果 `setTimeout()`在时间点 n 上被调用，那么执行定时器代码的 JavaScript 任务会在第 $n + 250$ 的时候加入 UI 队列。图 6-3 展示出示例中按钮被点击后所发生的事情与时间的关系。

请记住，定时器代码只有在创建它的函数执行完成之后，才有可能被执行。例如，如果前面代码中定时器延时变小，然后在创建定时器之后又调用另一个函数，那么定时器代码有可能在 `onclick` 事件处理完成之前加入队列：

```
var  button = document.getElementById("my-button");
button.onclick = function(){

    oneMethod();

    setTimeout(function(){
        document.getElementById("notice").style.color = "red";
    }, 50);

    anotherMethod();
};
```

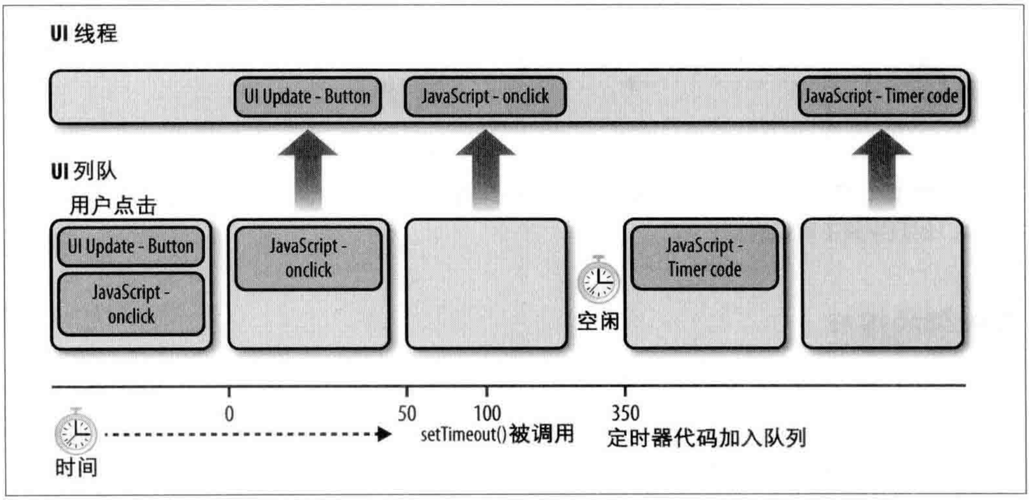

UI 线程
UI Update - Button
JavaScript - onclick
JavaScript - Timer code
UI 列队
用户点击
UI Update - Button
JavaScript - onclick
JavaScript - onclick
空闲
JavaScript - Timer code
0
50
100
setTimeout()被调用
350
定时器代码加入队列
时间

图 6-3　setTimeout()的第二个参数展示了新的 JavaScript 任务加入 UI 队列的时机

如果 `anotherMethod()` 的执行时间超过 50 毫秒，那么定时器代码已经在 `onclick` 处理完成前加入队列。这样做的影响是定时器代码会在 `onclick` 事件处理器执行完成后立刻运行，甚至觉察不出延迟。图 6-4 描绘了这种情况。

无论发生何种情况，创建一个定时器会造成 UI 线程暂停，如同它从一个任务切换到下一个任务。因此，定时器代码会重置所有相关的浏览器限制，包括长时间运行脚本定时器。此外，调用栈也在定时器的代码中重置为 0。这一特性使得定时器成为长时间运行 JavaScript 代码理想的跨浏览器解决方案。

提示：`setInterval()`函数和 `setTimeout()`几近相同，除了前者会重复添加 JavaScript 任务到 UI 队列。它们最主要的区别是，如果 UI 队列中已经存在由同一个 `setInterval()` 创建的任务，那么后续任务不会被添加到 UI 队列中。

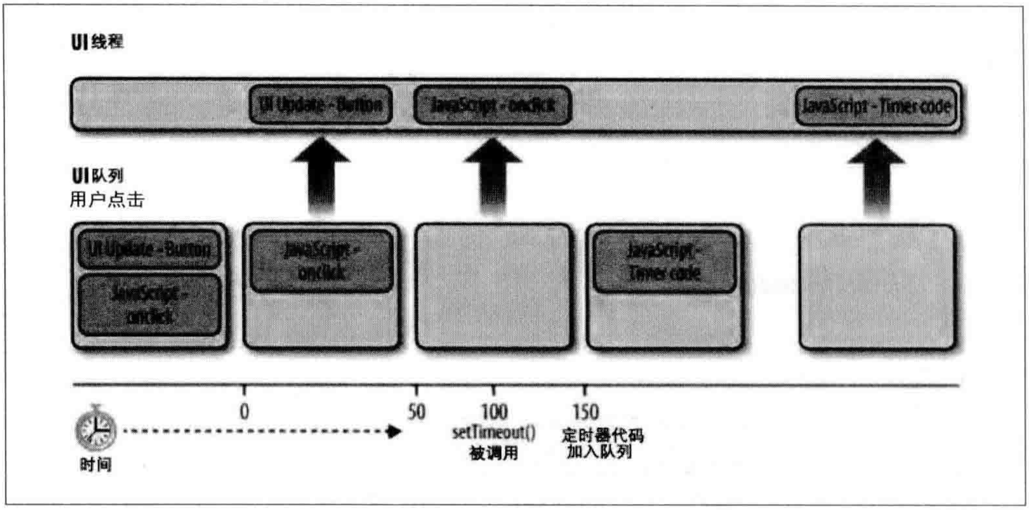

UI线程
UI Update - Button
JavaScript - onclick
JavaScript - Timer code
UI队列
用户点击
UI Update - Button
JavaScript - onclick
JavaScript - onclick
JavaScript - Timer code
0
50
100
setTimeout()
被调用
150
定时器代码
加入队列
时间

图 6-4 如果 setTimeout()中的函数需要消耗比定时器延时更长的运行时间，那么定时器代码中的延时几乎是不可见的。

定时器的精度

Timer Precision

JavaScript 定时器延迟通常不太精准，相差大约几毫秒。仅仅因为你指定定时器延时 250 毫秒，并不意味着任务一定会在调用 `setTimeout()` 之后过 250 毫秒时精确地加入队列。所有浏览器都试图尽可能准确，但通常会发生几毫秒偏移，或快或慢。正因如此，定时器不可用于测量实际时间。

在 Windows 系统中定时器分辨率为 15 毫秒，也就是说一个延时 15 毫秒的定时器将根据最后一次系统时间刷新而转换为 0 或 15。设置定时器延时小于 15 将会导致 IE 锁定，所以延迟的最小值建议为 25 毫秒（实际时间是 15 或 30）以确保至少有 15 毫秒延迟。

定时器延时的最小值有助于避免在其他浏览器和其他操作系统中的定时器出现分辨率问题。大多数浏览器在定时器延时等于或小于 10 毫秒时表现不太一致。

使用定时器处理数组

Array Processing with Timers

常见的一种造成长时间运行脚本的起因就是耗时过长的循环。如果你已经尝试了第四章中介绍的循环优化技术，但还是没能减少足够的运行时间，那么你下一步的优化步骤就是选用定时器。它的基本方法是把循环的工作分解到一系列定时器中。

典型的简单循环模式如下：

```
for  (var  i=0,  len=items.length;  i  <  len;  i++){
    process(items[i]);
}
```

这类循环结构运行时间过长的原因主要是 `process()`的复杂度或 `itemes` 的大小，或两者兼有。在我的那本“Professional JavaScript for Web Developer 2nd Edition”（Wrox 2009）中[译注2]，我列举了是否可以用定时器取代循环的两个决定性因素：

- 处理过程是否必须同步？
- 数据是否必须按顺序处理？

如果这两个问题的答案都是“否”，那么代码将适用于定时器分解任务。一种基本的异步代码模式如下：

```
var  todo = items.concat();   // 克隆原数组

setTimeout(function(){

    // 取得数组的下个元素并进行处理
    process(todo.shift());

    // 如果还有需要处理的元素，创建另一个定时器
    if(todo.length  >  0){
        setTimeout(arguments.callee, 25);
    }  else  {
        callback(items);
    }

}, 25);
```

这个模式的基本思路是创建一个原始数组的克隆，并将它作为数组项队列来处理。第一次调用 `setTimeout()` 创建一个定时器处理数组中的第一个条目。调用 `todo.shift()` 返回它的第一个条目然后把它从数组中删除。这个值作为参数传给 `process()`。处理完后，检查是否还有更多条目需要处理。如果 `todo` 数组中还有条目，那么就再启动一个定时器。因为下一个定时器需要运行相同的代码，所以第一个参数为 `arguments.callee`。该值指向当前正在运行的匿名函数。如果不再有条目需要处理，那么调用 `callback()` 函数。

提示：每个定时器的真实延时时间在很大程度上取决于具体情况。普遍来讲，最好使用至少 25 毫秒，因为再小的延时，对大多数 UI 更新来说不够用。

[译注2] 此书中文译名《JavaScript 高级程序设计：第二版》，已于 2010 年 7 月在国内发行。该书详情参见 *http://book.douban.com/subject/4886879/*。

这种模式与普通循环相比显然需要更多代码，所以可将该功能封装起来。例如：

```
function  processArray(items,  process,  callback){
    var  todo = items.concat();  // 克隆原数组

    setTimeout(function(){
        process(todo.shift());

        if  (todo.length  >  0){
            setTimeout(arguments.callee, 25);
        }  else  {
            callback(items);
        }

    }, 25);
}
```

processArray()函数用一种可重用的方式实现了前面的模式，它接收三个参数：待处理的数组，对每一个数组项调用的函数，处理完成后运行的回调函数。该函数的用法如下：

```
var  items = [123, 789, 323, 778, 232, 654, 219, 543, 321, 160];

function  outputValue(value){
    console.log(value);
}

processArray(items, outputValue, function(){
    console.log("Done!");
});
```

这段代码使用 processArray() 方法把数组的值输出到控制台，在所有处理完成后再打印一条消息。通过把定时器代码封装在一个函数里，它可在多处重用而无须多次实现。

提示：使用定时器处理数组的副作用是处理数组的总时长增加了。这是因为在每一个条目处理完成后 UI 线程会空闲出来，并且在下一条目开始处理之前会有一段延时。尽管如此，为避免锁定浏览器给用户带来的糟糕体验，这种取舍是有必要的。

分割任务

Splitting Up Tasks

我们通常会把一个任务分解成一系列子任务。如果一个函数运行时间太长，那么检查一下是否可以把它拆分为一系列能在较短时间内完成的子函数。往往可以把一行代码简单地看

成一个原子任务，即便是多行代码，也可以组合起来构成一个独立的任务。某些函数已经可以很容易基于函数调用进行拆分。例如：

```
function  saveDocument(id){

    // 保存文档
    openDocument(id)
    writeText(id);
    closeDocument(id);

    // 将成功信息更新至界面
    updateUI(id);
}
```

如果这个函数运行时间太长，可以很容易地把它拆分成一系列更小的步骤，把每个独立的方法放在定时器中调用。你可以将每个函数都放入一个数组，然后使用前一小节提到的数组处理模式：

```
function saveDocument(id){

    var tasks = [openDocument, writeText, closeDocument, updateUI];

    setTimeout(function(){

        // 执行下一个任务
        var task = tasks.shift();
        task(id);

        // 检查是否还有其他任务
        if (tasks.length > 0){
            setTimeout(arguments.callee,  25);
        }
    }, 25);
}
```

这个版本的函数把每个方法都放入 tasks 数组，然后每次在定时器中只调用一个方法。本质上，现在变成了数组处理模式，唯一的区别在于处理数组条目时调用的函数就包含在条目中。正如上一章中讨论的，这个模式可以进行封装以备复用：

```
function multistep(steps, args, callback){

    var tasks = steps.concat();     // 克隆数组

    setTimeout(function(){

        // 执行下一个任务
        var task = tasks.shift();
        task.apply(null, args || []);

        // 检查是否还有其他任务
```

```
            if (tasks.length > 0){
                setTimeout(arguments.callee, 25);
            } else {
                callback();
            }
        }, 25);
    }
```

multistep()函数接收三个参数：由待执行函数组成的数组，为每个函数运行时提供参数的数组，以及处理结束时调用的回调函数。函数用法如下：

```
    function saveDocument(id){

        var tasks = [openDocument, writeText, closeDocument, updateUI];
        multistep(tasks, [id], function(){
            alert("Save completed!");
        });
    }
```

请注意 multistep()的第二个参数必须为数组，它创建时只包含一个 id。正如数组处理那样，使用此函数的前提条件是：任务可以异步处理而不影响用户体验或造成相关代码错误。

记录代码运行时间

Timed Code

有时每次只执行一个任务的效率不高。考虑这种情况：如处理一个长度为 1000 项的数组，每处理一项需时 1 毫秒。如果每个定时器中只处理一项，且在两次处理之间产生 25 毫秒的延迟，这意味着处理数组需要的总时间为 (25 + 1) × 1 000 = 26 000 毫秒（或 26 秒）。如果一次批处理 50 个，每批之间有 25 毫秒延迟呢？整个处理过程时间变成 (1 000 / 50) × 25 + 1 000 = 1 500 毫秒（或 1.5 秒），而且用户不会觉察到界面阻塞，因为最长的脚本运行只持续了 50 毫秒。通常来说批量处理比单个处理要快。

如果你还记得 JavaScript 可以持续运行的最长时间为 100 毫秒，那么你可以优化先前的模式。我的建议是把这个数字减半，不要让任何 JavaScript 代码持续运行 50 毫秒以上，这样做只是确保代码永远不会影响用户体验。

可以通过原生的 Date 对象来跟踪代码的运行时间。这是大多数 JavaScript 分析工具的工作原理：

```
    var start = +new Date(),
        stop;

    someLongProcess();
```

```
stop = +new Date();

if(stop-start < 50){
    alert("Just about right.");
} else {
    alert("Taking too long.");
}
```

由于每个新创建的 `Date` 对象以当前系统时间初始化，你可以定时创建 `Date` 对象并比较它们的值来记录代码运行时间。加号（+）可以将 `Date` 对象转化成数字，那么在后续的数学运算中就无须转换了。这种技术也可以用来优化之前的定时器模式。

通过添加一个时间检测机制来改进 `processArray()` 方法，使得每个定时器能处理多个数组条目：

```
function timedProcessArray(items, process, callback){
    var todo = items.concat();  // 克隆原始数组

    setTimeout(function(){
        var start = +new Date();

        do {
            process(todo.shift());
        } while (todo.length > 0 && (+new Date() - start < 50));

        if (todo.length > 0){
            setTimeout(arguments.callee, 25);
        } else {
            callback(items);
        }

    }, 25);
}
```

该函数中添加了一个 `do-while` 循环，它在每个数组条目处理完后检测执行时间。定时器函数执行时数组中始终会包含至少一个条目，因此后测循环比前测循环更为合理。在 Firefox 3 中，如果 `process()`是个空函数，处理 1000 项的数组需要 38~43 毫秒；原始的 `processArray()` 函数处理相同数组需要超过 25 000 毫秒。这就是定时任务的作用，能避免把任务分解成过于零碎的片断。

定时器与性能

Timers and Performance

定时器会让你的 JavaScript 代码整体性能发生翻天覆地的变化，但过度使用也会对性能造成负面影响。本节中的代码使用了定时器序列，同一时间只有一个定时器存在，只有当这个定时器结束时才会新创建一个。通过这种方法使用定时器不会导致性能问题。

当多个重复的定时器同时创建往往会出现性能问题。因为只有一个 UI 线程，而所有的定时器都在争夺运行时间。Google Mobile 的 Neil Thomas 将此问题作为度量 iPhone 和 Android 平台上移动版 Gmail 应用性能的方法进行了研究[注4]。

Thomas 发现那些间隔在 1 秒或 1 秒以上的低频率的重复定时器几乎不会影响 Web 应用的响应速度。这种情况下定时器延迟远远超过 UI 线程产生瓶颈的值，因此可安全地重复使用。当多个重复定时器使用较高的频率（100 到 200 毫秒之间）时，Thomas 发现移动版 Gmail 应用会明显变慢，响应也不及时。

Thomas 研究的言外之意是，在你的 Web 应用中限制高频率重复定时器的数量。作为替代方案，Thomas 建议创建一个独立的重复定时器，每次执行多个操作。

Web Workers

自 JavaScript 诞生以来，还没有办法在浏览器 UI 线程之外运行代码。Web Workers API 改变了这种状况，它引入了一个接口，能使代码运行且不占用浏览器 UI 线程的时间。作为 HTML5 最初的一部分，Web Workers API 已经被分离出去成为独立的规范（*http://www.w3.org/TR/workers/*）。Web Workers 已经被 Firefox 3.5、Chrome 3 和 Safari 4 原生支持。

Web workers 给 Web 应用带来潜在的巨大性能提升，因为每个新的 Worker 都在自己的线程中运行代码。这意味着 Worker 运行代码不仅不会影响浏览器 UI，也不会影响其他 Worker 中运行的代码。

Worker 运行环境

Worker Environment

由于 Web Workers 没有绑定 UI 线程，这也意味着它们不能访问浏览器的许多资源。JavaScript 和 UI 共享同一进程的部分原因是它们之间互相访问频繁，因此这些任务失控会导致糟糕的用户体验。Web Workers 从外部线程中修改 DOM 会导致用户界面出现错误，但是每个 Web Worker 都有自己的全局运行环境，其功能只是 JavaScript 特性的一个子集。Worker 运行环境由如下部分组成。

- 一个 `navigator` 对象，只包括四个属性：`appName`、`appVersion`、user `Agent` 和 `platform`。

注 4：　全文位于 *http://googlecode.blogspot.com/2009/07/gmail-for-mobile-html5-series-using.html*。

- 一个 location 对象（与 window.location 相同，不过所有属性都是只读的）。
- 一个 self 对象，指向全局 worker 对象。
- 一个 importScripts() 方法，用来加载 Worker 所用到的外部 JavaScript 文件。
- 所有的 ECMAScript 对象，诸如：Object、Array、Date 等。
- XMLHttpRequest 构造器。
- setTimeout()和 setInterval()方法。
- 一个 close()方法，它能立刻停止 Worker 运行。

由于 Web Worker 有着不同的全局运行环境，因此你无法从 JavaScript 代码中创建它。事实上，你需要创建一个完全独立的 JavaScript 文件，其中包含了需要在 Worker 中运行的代码。要创建网页工人线程，你必须传入这个 JavaScript 文件的 URL：

```
var worker = new Worker("code.js");
```

此代码一旦执行，将为这个文件创建一个新的线程和一个新的 Worker 运行环境。该文件会被异步下载，直到文件下载并执行完成后才会启动此 Worker。

与 Worker 通信

Worker Communication

Worker 与网页代码通过事件接口进行通信。网页代码可以通过 postMessage() 方法给 Worker 传递数据，它接收一个参数，即需要传递给 Worker 的数据。此外， Worker 还有一个用来接收信息的 onmessage 事件处理器。例如：

```
var worker = new Worker("code.js");
worker.onmessage = function(event){
    alert(event.data);
};
worker.postMessage("Nicholas");
```

Worker 通过触发 message 事件来接收数据。定义 onmessage 事件处理器后，该事件对象就具有一个 data 属性用于存放传入的数据。Worker 可通过它自己的 postMessage() 方法把信息回传给页面：

```
// code.js 内部代码
self.onmessage = function(event){
    self.postMessage("Hello, " + event.data + "!");
};
```

最终的字符串在 Worker 的 onmessage 事件处理器中构造完成。消息系统是网页和 Worker 通信的唯一途径。

只有特定类型的数据可以使用 `postMessage()` 传递。你可以传递原始值（字符串、数字、布尔值、`null` 和 `undefined`），也可以传递 `Object` 和 `Array` 的实例，其他类型就不允许了。有效数据会被序列化，传入或传出 Worker，然后反序列化。虽然看上去对象可以直接传入，但对象实例完全是相同数据的独立表述。尝试传入一个不支持的数据类型会导致 JavaScript 错误。

提示：Safari 4 的实现的 Worker 只允许 `postMessage()` 传递字符串。自从 Firefox 3.5 实现的 Worker 允许传递序列化数据后，相应的规范也进行了更新。

加载外部文件

Loading External Files

Worker 通过 `importScripts()` 方法加载外部 JavaScript 文件，该方法接收一个或多个 JavaScript 文件 URL 作为参数。`importScripts()` 的调用过程是阻塞式的，直到所有文件加载并执行完成之后，脚本才会继续运行。由于 Worker 在 UI 线程之外运行，所以这种阻塞并不会影响 UI 响应。例如：

```
// code.js 内部代码
importScripts("file1.js", "file2.js");

self.onmessage = function(event){
    self.postMessage("Hello, " + event.data + "!");
};
```

这段代码第一行中引入了两个 JavaScript 文件，它们将在 Worker 线程的上下文中用到。

实际应用

Practical Uses

Web Workers 适用于那些处理纯数据，或者与浏览器 UI 无关的长时间运行脚本。尽管它看上去用处不大，但 Web 应用中通常有一些数据处理功能将受益于 Worker 而不是定时器。

考虑这样一个例子：解析一个很大的 JSON 字符串（JSON 解析将会在第 7 章讨论）。假设数据量足够大，至少需要 500 毫秒才能完成解析。很明显时间太长了，超出了客户端允许 JavaScript 运行的时间，因为它会干扰用户体验。而此任务难以分解成若干个使用定时器的小任务，因此 Worker 成为最理想的解决方案。下面的代码描绘了它在网页中的使用：

```
var worker = new Worker("jsonparser.js");

//数据就位时，调用事件处理器
worker.onmessage = function(event){

    //JSON 结构被回传回来
    var jsonData = event.data;

    //使用 JSON 结构
    evaluateData(jsonData);
};

//传入要解析的大段 JSON 字符串
worker.postMessage(jsonText);
```

Worker 中负责解析 JSON 的代码如下：

```
//jsonparser.js 内部代码

//当 JSON 数据存在时，该事件处理器会被调用
self.onmessage = function(event){

    //JSON 字符串由 event.data 传入
    var jsonText = event.data;

    //解析
    var jsonData = JSON.parse(jsonText);

    //回传结果
    self.postMessage(jsonData);
};
```

请注意，即使 `JSON.parse()` 需要执行 500 毫秒或更长时间，也无须写任何额外的代码来分解处理过程。该执行过程发生在一个独立的线程中，因此你可以让它一直运行直到解析完成而不会干扰用户体验。

页面中使用 `postMessage()` 方法向 Worker 传递 JSON 字符串。Worker 在它自己的 `onmessage` 事件处理器中接收此字符串，即 `event.data`，然后开始解析。完成后所产生的 JSON 对象由 Worker 的 `postMessage()` 方法传回页面。此时该对象即为页面的 `onmessage` 事件处理器的 `event.data`。请记住，这段代码目前只能运行在 Firefox 3.5 及更高版本中，因为 Safari 4 和 Chrome 3 的实现只允许在页面和 Worker 之间传递字符串。

解析一个大字符串只是许多受益于 Web Workers 的任务之一。其他可能受益的任务如下：

- 编码/解码大字符串。
- 复杂数学运算（包括图像或视频处理）。
- 大数组排序。

任何超过 100 毫秒的处理过程，都应当考虑 Worker 方案是不是比基于定时器的方案更为合适。当然，前提是浏览器支持 Web Workers。

小结

Summary

JavaScript 和用户界面更新在同一个进程中运行，因此一次只能处理一件事情。这意味着当 JavaScript 代码正在运行时，用户界面不能响应输入，反之亦然。高效地管理 UI 线程就是要确保 JavaScript 不能运行太长时间，以免影响用户体验。最后，请牢记如下几点：

- 任何 JavaScript 任务都不应当执行超过 100 毫秒。过长的运行时间会导致 UI 更新出现明显的延迟，从而对用户体验产生负面影响。
- JavaScript 运行期间，浏览器响应用户交互的行为存在差异。无论如何，JavaScript 长时间运行将导致用户体验变得混乱和脱节。
- 定时器可用来安排代码延迟执行，它使得你可以把长时间运行脚本分解成一系列的小任务。
- Web Workers 是新版浏览器支持的特性，它允许你在 UI 线程外部执行 JavaScript 代码，从而避免锁定 UI。

Web 应用越复杂，积极主动地管理 UI 线程就越重要。即使 JavaScript 代码再重要，也不应该影响用户体验。

第 7 章

Ajax

Ross Harmes

Ajax 是高性能 JavaScript 的基础。它可以通过延迟下载体积较大的资源文件来使得页面加载速度更快。它通过异步的方式在客户端和服务端之间传输数据，从而避免页面资源一窝蜂地下载。它甚至可以只用一个 HTTP 请求就获取整个页面的资源。选择适合的传输方式和最有效的数据格式，可以显著改善用户和网站的交互体验。

本章会介绍从服务器收发数据速度最快的技术，以及最为有效的数据编码格式。

数据传输
Data Transmission

Ajax，从最基本的层面来说，是一种与服务器通信而无须重载页面的方法；数据可以从服务器获取或发送给服务器。有多种不同的方法建立这种通信通道，每种方法都有各自的优点和限制。本节简要地介绍这些方法，并讨论每一种方法对性能的影响。

请求数据
Requesting Data

有五种常用技术用于向服务器请求数据：

- XMLHttpRequest (XHR)
- Dynamic script tag insertion 动态脚本注入
- iframes
- Comet
- Multipart XHR

在现代高性能 JavaScript 中使用的三种技术是：XHR、动态脚本注入和 multipart XHR。Comet 和 iframes（作为数据传输技术）往往用在极端情况下，在这里不作讨论。

XMLHttpRequest

XMLHttpRequest（XHR）是目前最常用的技术，它允许异步发送和接收数据。所有的主流浏览器对它都提供了完善的支持，而且它还能精确地控制发送请求和数据接收。你可以在请求中添加任何头信息和参数（包括 GET 和 POST），并读取服务器返回的所有头信息，以及响应文本。接下来是一个使用范例：

```
var url = '/data.php';
var params = [
    'id=934875',
    'limit=20'
];

var req = new XMLHttpRequest();

req.onreadystatechange = function() {
    if (req.readyState === 4) {
        var responseHeaders = req.getAllResponseHeaders(); // 获取响应头信息
        var data = req.responseText; // 获取数据
        //数据处理
    }
}

req.open('GET', url + '?' + params.join('&'), true);
req.setRequestHeader('X-Requested-With', 'XMLHttpRequest'); // 设置请求头信息
req.send(null); // 发送一个请求
```

这个例子显示了如何从带有参数的 URL 请求数据，以及如何读取响应文本和头信息。`readyState` 值等于 4 表示整个响应已接收完毕，可进行操作。

通过监听 `readyState` 值等于 3，你可以与正在传输中的服务器响应进行交互，可以通过 `readyState` 值等于 3 说明此时正在与服务器交互，响应信息还在传输过程中。这就是所谓的"流"（streaming），它是提升数据请求性能的强大工具：

```
req.onreadystatechange = function() {

    if (req.readyState === 3) { // 接收到部分信息，但不是所有
        var dataSoFar = req.responseText;
        ...
    }
    else if (req.readyState === 4) { // 所有信息接收完毕
        var data = req.responseText;
        ...
        }
    }
```

由于 XHR 提供了高级的控制，所以浏览器对其增加了一些限制。你不能使用 XHR 从外域请求数据，而且低版本的 IE 不仅不支持“流”，也不会提供 `readyState` 为 3 的状态。从服务器传回的数据被当作字符串或者 XML 对象，这意味着处理大量数据将会很慢。

尽管有这些缺点，XHR 依然是最常用也最强大的数据请求技术，它应当成为你的首选。

使用 XHR 时，POST 和 GET 的对比。当使用 XHR 请求数据时，你需要在 POST 和 GET 之间做出选择。对于那些不会改变服务器状态，只会获取数据（这被称为“幂等行为[译注1]”）的请求，应该使用 GET。经 GET 请求的数据会被缓存起来，如果需要多次请求同一数据的话，它会有助于提升性能。

只有当请求的 URL 加上参数的长度接近或超过 2 048 个字符时，才应该用 POST 获取数据。这是因为 IE 限制 URL 长度，过长时将会导致请求的 URL 被截断。

动态脚本注入

这种技术克服了 XHR 的最大限制：它能跨域请求数据。这是一个 Hack，你不需要实例化一个专用对象，而可使用 JavaScript 创建一个新的脚本标签，并设置它的 `src` 属性为不同域的 URL。

```
var scriptElement = document.createElement('script');
scriptElement.src = 'http://any-domain.com/javascript/lib.js';
document.getElementsByTagName('head')[0].appendChild(scriptElement);
```

但是与 XHR 相比，动态脚本注入提供的控制是有限的。你不能设置请求的头信息。参数传递也只能使用 GET 方式，而不是 POST 方式。你不能设置请求的超时处理或重试；事实上，就算失败了也不一定知道。你必须等待所有数据都已返回，才可以访问它们。你不能访问请求的头信息，也不能把整个响应消息作为字符串来处理。

最后一点特别重要。因为响应消息作为脚本标签的源码，它必须是可执行的 JavaScript 代码。你不能使用纯 XML、纯 JSON 或其他任何格式的数据，无论哪种格式，都必须封装在一个回调函数中。

```
var scriptElement = document.createElement('script');
scriptElement.src = 'http://any-domain.com/javascript/lib.js';
document.getElementsByTagName('head')[0].appendChild(scriptElement);

function jsonCallback(jsonString) {
    var data = eval('(' + jsonString + ')');
    // 处理数据…
}
```

[译注1] 根据 HTTP 协议：假如在不考虑诸如错误或者过期等问题的情况下，若干次请求的副作用与单次请求相同或者根本没有副作用，那么这些请求方法就能够被视作“幂等”的。GET 请求对服务器不产生其他副作用，所以具有幂等属性。详细解释请参阅维基百科条目 *http://zh.wikipedia.org/zh-cn/*超文本传输协议#.E5.B9.82.E7.AD.89.E6.96.B9.E6.B3.95。

在本例中，lib.js 文件需要把数据封装在 `jsonCallback` 函数里：

```
jsonCallback({ "status": 1, "colors": [ "#fff", "#000", "#ff0000" ] });
```

尽管有这些限制，这项技术的速度却非常快。响应消息是作为 JavaScript 执行，而不是作为字符串需要进一步处理。正因为如此，它有潜力成为客户端获取并解析数据最快的方法。我们对比了动态脚本注入和 XHR 的性能，将在本章后续的 JSON 一节中讨论。

使用这种技术从那些你无法直接控制的服务器上请求数据时需要小心。JavaScript 没有任何权限和访问控制的概念，因此你使用动态脚本注入添加到页面中的任何代码都可以完全控制整个页面。包括修改任何内容，把用户重定向到其他网站，甚至跟踪用户在页面上的操作并发送数据到第三方服务器。引入外部来源的代码时请务必多加小心。

Multipart XHR

现在提到的是一项最新的技术，multipart XHR（MXHR）允许客户端只用一个 HTTP 请求就可以从服务端向客户端传送多个资源。它通过在服务端将资源（CSS 文件、HTML 片段、JavaScript 代码或 base64 编码的图片）打包成一个由双方约定的字符串分割的长字符串并发送到客户端。然后用 JavaScript 代码处理这个长字符串，并根据它的 mime-type 类型和传入的其他“头信息”解析出每个资源。

让我们从头至尾观察该过程。首先，一个用来获取多张图片的请求发送到服务器：

```
var req = new XMLHttpRequest();

req.open('GET', 'rollup_images.php', true);
req.onreadystatechange = function() {
    if (req.readyState == 4) {
        splitImages(req.responseText);
    }
};
req.send(null);
```

这是一个非常简单的请求。你向 `rollup_images.php` 请求数据，一旦接收到返回结果，就把它交给函数 `splitImages` 处理。

下一步，服务器读取图片并将它们转换为字符串：

```
// 读取图片并将它们转换为 base64 编码的字符串

$images = array('kitten.jpg', 'sunset.jpg', 'baby.jpg');
foreach ($images as $image) {

    $image_fh = fopen($image, 'r');
    $image_data = fread($image_fh, filesize($image));
```

```
        fclose($image_fh);
            $payloads[] = base64_encode($image_data);
        }
    }

    // 把字符串合并成一个长字符串，然后输出它

    $newline = chr(1); // 该字符串不会出现在任何 base64 字符串中

    echo implode($newline, $payloads);
```

这段 PHP 代码读取了三张图片，并把它们转换为 base64 编码的长字符串。它们之间用一个简单的 Unicode 编码的字符 1 连接，然后返回给客户端。

一旦数据到达客户端，该数据由 `splitImages` 函数处理：

```
function splitImages(imageString) {

    var imageData = imageString.split("\u0001");
    var imageElement;

    for (var i = 0, len = imageData.length; i < len; i++) {

        imageElement = document.createElement('img');
        imageElement.src = 'data:image/jpeg;base64,' + imageData[i];
        document.getElementById('container').appendChild(imageElement);
    }
}
```

此函数将连接后的字符串分解成三段。每一段用来创建一个图片元素，然后将图片元素插入页面中。图片不是由 base64 字符串转换成二进制，而是使用 `data:` URL 的方式创建，并指定 mime-type 为 `images/jpeg`。

最终的结果是：在一次 HTTP 请求中向浏览器传送了三张图片。你也可以传送 20 张或 100 张图片，这样的话响应消息会更大，但仍然只用了一次 HTTP 请求。当然，这种方式也可以扩展到其他资源的类型。JavaScript 文件、CSS 文件、HTML 片段以及多种类型的图片都能合并成一次响应。任何数据类型都可以被 JavaScript 作为字符串发送。下面的函数用于将 JavaScript 代码、CSS 样式和图片转换为浏览器可用的资源：

```
function handleImageData(data, mimeType) {
    var img = document.createElement('img');
    img.src = 'data:' + mimeType + ';base64,' + data;
    return img;
}

function handleCss(data) {
    var style = document.createElement('style');
    style.type = 'text/css';

    var node = document.createTextNode(data);
    style.appendChild(node);
    document.getElementsByTagName('head')[0].appendChild(style);
}

function handleJavaScript(data) {
    eval(data);
}
```

由于 MXHR 响应消息的体积越来越大，因此有必要在每个资源收到时就立刻处理，而不是等到整个响应消息接收完成。这可以通过监听 `readyState` 为 3 的状态来实现：

```
var req = new XMLHttpRequest();
var getLatestPacketInterval, lastLength = 0;

req.open('GET', 'rollup_images.php', true);
req.onreadystatechange = readyStateHandler;
req.send(null);

function readyStateHandler{
    if (req.readyState === 3 && getLatestPacketInterval === null) {

        // 开始轮询

        getLatestPacketInterval = window.setInterval(function() {
            getLatestPacket();
        }, 15);
    }

    if (req.readyState === 4) {

        // 停止轮询

        clearInterval(getLatestPacketInterval);

        // 获取最后一个数据包

        getLatestPacket();
    }
}

function getLatestPacket() {
    var length = req.responseText.length;
    var packet = req.responseText.substring(lastLength, length);

    processPacket(packet);
    lastLength = length;
}
```

当 `readyState` 为 3 的状态第一次触发时，启动一个定时器，每隔 15 毫秒检查一次响应中的新数据。数据片段会被收集起来，直到发现一个分隔符，然后就把遇到分隔符之前收集的所有数据作为一个完整的资源进行处理。

编写健壮的 MXHR 代码很复杂，但值得深入研究。访问以下网址可获取完整的类库：*http://techfoolery.com/mxhr/*。

使用这个技术有一些缺点，其中最大的缺点是以这种方式获得的资源不能被浏览器缓存。如果你使用 MXHR 获取一个特定的 CSS 文件，然后在下一个页面中正常加载它，它将不会存在于缓存里。这是因为合并后的资源是作为字符串传输的，然后被 JavaScript 代码分解成片段。由于无法用编程的方式向浏览器缓存里注入文件，因此用这种方式获取的资源无法被缓存。

另一个缺点是，老版本的 IE 不支持 `readyState` 为 3 的状态和 `data:` URL。IE 8 倒是两个都支持，但在 IE 6 和 IE 7 中必须设法变通。

尽管有这些缺点，但某些情况下 MXHR 依然能显著提升页面的整体性能：

- 页面包含了大量其他地方用不到的资源（因此也无须缓存），尤其是图片。
- 网站已经在每个页面中使用一个独立打包的 JavaScript 或 CSS 文件以减少 HTTP 请求；因为对每个页面来说这些文件都是唯一的，所以并不需要从缓存中读取数据，除非重载页面。

由于 HTTP 请求是 Ajax 中最大的瓶颈之一，因此减少其需要的数量会对整个页面的性能有很大的影响。尤其是当你将 100 个图片请求转化为一个 MXHR 请求时。专门有一个这样的测试，它在各种主流浏览器上传送大量图片，结果显示此技术比各自独立请求的方式快 4 到 10 倍。你也可以自己运行该测试：*http://techfoolery.com/mxhr/*。

发送数据

Sending Data

有时候你并不关心接收数据，而只需要将数据发送给服务器。你可以发送不涉及隐私的用户信息以备日后分析，或者你也可以捕获所有脚本错误并把细节发送到服务器进行记录和报警。当数据只需要发送到服务器时，有两种广泛使用的技术：XHR 和信标（beacons）。

XMLHttpRequest

虽然 XHR 主要用于从服务器获取数据，但它同样能用于把数据传回服务器。数据可以用 GET 或 POST 的方式传回来，包括任意数量的 HTTP 头信息。这给你很大的灵活性。当你要传回的数据超出浏览器的最大 URL 尺寸限制时 XHR 特别有用。这种情况下，你可以用 POST 方式回传数据：

```
var url = '/data.php';
var params = [
    'id=934875',
    'limit=20'
];
```

```
var req = new XMLHttpRequest();

req.onerror = function() {
    // 出错
};

req.onreadystatechange = function() {
    if (req.readyState == 4) {
        // 成功
    }
};

req.open('POST', url, true);
req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
req.setRequestHeader('Content-Length', params.length);
req.send(params.join('&'));
```

正如你在这个例中看到的，我们并不处理发送失败的情况。当用 XHR 来捕获用户统计信息时这么做没什么问题，但如果发送到服务器的是至关紧要的数据，你可以增加一些代码来实现在失败时重试：

```
function xhrPost(url, params, callback) {

    var req = new XMLHttpRequest();

    req.onerror = function() {
        setTimeout(function() {
            xhrPost(url, params, callback);
        }, 1000);
    };

    req.onreadystatechange = function() {
        if (req.readyState == 4) {
            if (callback && typeof callback === 'function') {
                callback();
            }
        }
    };

    req.open('POST', url, true);
    req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
    req.setRequestHeader('Content-Length', params.length);
    req.send(params.join('&'));
}
```

当使用 XHR 发送数据到服务器时，GET 方式会更快。这是因为，对于少量数据而言，一个 GET 请求往服务器只发送一个数据包。而一个 POST 请求，至少要发送两个数据包，一个装载头信息，另一个装载 POST 正文。POST 更适合发送大量数据到服务器，因为它不关心额外数据包的数量，另一个原因是 IE 对 URL 长度有限制，它不可能使用过长的 GET 请求。

Beacons

这项技术非常类似动态脚本注入。使用 JavaScript 创建一个新的 `Image` 对象，并把 `src` 属性设置为服务器上脚本的 URL。该 URL 包含了我们要通过 GET 传回的键值对数据。请注意并没有创建 `img` 元素或把它插入 DOM。

```
var url = '/status_tracker.php';
var params = [
    'step=2',
    'time=1248027314'
];

(new Image()).src = url + '?' + params.join('&');
```

服务器会接收到数据并保存下来，它无须向客户端发送任何回馈信息，因此没有图片会实际显示出来。这是给服务器回传信息最有效的方式。它的性能消耗很小，而且服务端的错误完全不会影响到客户端。

图片信标很简单，但也意味着它能做的事情是有限的。你无法发送 POST 数据，而 URL 的长度有最大值，所以你可以发送的数据的长度被限制得相当小。你可以接收服务器返回的数据，但只局限于非常少的几种方式。一种方式是监听 `Image` 对象的 `load` 事件，它会告诉你服务器是否成功接收数据。你还可以检查服务器返回的图片的宽度和高度（如果返回的是图片的话）并使用这些数字通知你服务器的状态。举个例子，宽度为 1 表示“成功”，为 2 表示“重试”。

如果你不需要在响应中返回数据，就应该发送一个不带消息正文的 `204 No Content` 状态码，它将阻止客户端继续等待永远不会到来的消息正文：

```
var url = '/status_tracker.php';
var params = [
    'step=2',
    'time=1248027314'
];

var beacon = new Image();
beacon.src = url + '?' + params.join('&');

beacon.onload = function() {
    if (this.width == 1) {
        // 成功
    }
    else if (this.width == 2) {
        // 失败，请重试并创建另一个信标
    }
};

beacon.onerror = function() {
        // 出错，稍候重试并创建另一个信标。
    };
```

信标是向服务器回传数据最快且最有效的方式。服务器根本不需要发送任何响应正文，因此你也无须担心客户端下载数据。唯一的缺点是你能接收到的响应类型是有限的。如果你需要返回大量数据给客户端，那么请使用 XHR。如果你只关心发送数据到服务器（可能需要极少的返回信息），那么请使用图片信标。

数据格式

Data Formats

当考虑数据传输技术时，你必须考虑这些因素：功能集、兼容性、性能以及方向（发送给服务器还是从服务器接收）。当考虑数据格式时，唯一需要比较的标准就是速度。

没有哪种数据格式会始终比其他格式更好。优劣取决于要传输的数据以及它在页面上的用途，一种可能下载更快，而另一种可能解析更快。在本节中，我们创建了一个挂件（widget）用于搜索用户信息，并使用四种主流的数据格式实现它。这要求我们在服务端格式化一个用户列表，将它返回给浏览器，接着把列表解析成原生的 JavaScript 数据结构，然后搜索给定的字符串。每种数据格式都将在列表文件尺寸、解析速度以及在服务器上构造的难易程度几方面进行比较。

XML

当 Ajax 最先开始流行时，它选择了 XML 作为数据格式。当时它有很多的优势：极佳的通用性（服务端和客户端都能完美支持）、格式严格，且易于验证。那时 JSON 还没有正式作为交换格式，几乎所有服务端语言都有操作 XML 的类库。

下面是一个 XML 格式的用户列表示例：

```
    <?xml version="1.0" encoding='UTF-8'?>
    <users total="4">
        <user id="1">
            <username>alice</username>
            <realname>Alice Smith</realname>
            <email>alice@alicesmith.com</email>
        </user>
        <user id="2">
            <username>bob</username>
            <realname>Bob Jones</realname>
            <email>bob@bobjones.com</email>
        </user>
        <user id="3">
            <username>carol</username>
        <realname>Carol Williams</realname>
        <email>carol@carolwilliams.com</email>
    </user>
    <user id="4">
        <username>dave</username>
        <realname>Dave Johnson</realname>
        <email>dave@davejohnson.com</email>
    </user>
</users>
```

相比其他格式，XML 极其冗长。每个单独的数据片段都依赖大量结构，所以有效数据的比例非常低。而且 XML 的语法有些模糊。当把一个数据结构转换成 XML 时，你应该把对象参数放到对象元素的属性中，还是放在独立的子元素中呢？你应该使用描述清晰的长标签名还是高效但难以辨认的短标签名呢？语法的解析过程同样含混，你必须提前知道 XML 响应的布局，然后才能弄清楚它的含义。

一般情况下，解析 XML 需要占用 JavaScript 程序员相当一部分的精力。除了要提前知道详细结构之外，还必须确切地知道如何解析这个结构并费力地将它重组到 JavaScript 对象中。这既非易事也不能自动完成，不像其他三种数据结构那样。

下面的例子演示了如何把特定 XML 响应解析成一个对象：

```
function parseXML(responseXML) {

    var users = [];
    var userNodes = responseXML.getElementsByTagName('users');
    var node, usernameNodes, usernameNode, username,
        realnameNodes, realnameNode, realname,
        emailNodes, emailNode, email;

    for (var i = 0, len = userNodes.length; i < len; i++) {

        node = userNodes[i];
        username = realname = email = '';

        usernameNodes = node.getElementsByTagName('username');
        if (usernameNodes && usernameNodes[0]) {
            usernameNode = usernameNodes[0];
            username = (usernameNodes.firstChild) ?
                usernameNodes.firstChild.nodeValue : '';
        }

        realnameNodes = node.getElementsByTagName('realname');
        if (realnameNodes && realnameNodes[0]) {
            realnameNode = realnameNodes[0];
            realname = (realnameNodes.firstChild) ?
                realnameNodes.firstChild.nodeValue : '';
        }

        emailNodes = node.getElementsByTagName('email');
```

```
            if (emailNodes && emailNodes[0]) {
                emailNode = emailNodes[0];
                email = (emailNodes.firstChild) ?
                    emailNodes.firstChild.nodeValue : '';
            }

            users[i] = {
                id: node.getAttribute('id'),
                username: username,
                realname: realname,
                email: email
            };
        }

        return users;
    }
```

正如你所看到的，它需要先检查每一个标签，确保存在时才读取它的值。这在很大程度上依赖于 XML 的结构。

一个更有效的方法是把每一个值都转换为 <user> 标签的属性。这样一来，还是同样的数据，但文件尺寸变小了。下面是一个把值转成属性的用户列表的例子：

```
<?xml version="1.0" encoding='UTF-8'?>
<users total="4">
    <user id="1-id001" username="alice" realname="Alice Smith"
        email="alice@alicesmith.com" />
    <user id="2-id001" username="bob" realname="Bob Jones"
        email="bob@bobjones.com" />
    <user id="3-id001" username="carol" realname="Carol Williams"
        email="carol@carolwilliams.com" />
    <user id="4-id001" username="dave" realname="Dave Johnson"
        email="dave@davejohnson.com" />
</users>
```

解析这个简化后的 XML 响应要容易得多：

```
function parseXML(responseXML) {

    var users = [];
    var userNodes = responseXML.getElementsByTagName('users');

    for (var i = 0, len = userNodes.length; i < len; i++) {
        users[i] = {
            id: userNodes[i].getAttribute('id'),
            username: userNodes[i].getAttribute('username'),
            realname: userNodes[i].getAttribute('realname'),
            email: userNodes[i].getAttribute('email')
        };
    }

    return users;
}
```

XPath

虽然它超出了本章内容的范围，但说明 XPath 在解析 XML 文档时比 `getElementsByTagName` 快许多。需要注意的是，它并未得到广泛支持。因此你必须使用旧式 DOM 遍历方法编写降级的代码。现在，DOM Level 3 的 XPath 已经由 Firefox、Safari、Chrome 和 Opera 实现。IE 8 有类似的接口，但稍微高级一些。[译注2]

响应内容大小和解析事件

让我们看看下表中的 XML 性能数据。

格式	大小	下载耗时	解析耗时	总耗时
标准的 XML	582 960 字节	999.4 毫秒	343.1 毫秒	1342.5 毫秒
简化的 XML	437 960 字节	475.1 毫秒	83.1 毫秒	558.2 毫秒

提示：每种数据类型分别通过长度为 100、500、1 000 和 5 000 的用户列表进行测试。每个列表在同一个浏览器中下载并解析 10 次，分别计算下载耗时、解析耗时和文件尺寸的平均值。所有数据格式和传输技术的完整测试结果，请访问 *http://techfoolery.com/formats/*，你也可以从中下载代码到本地自行测试。

正如你所看到的，相比使用子标签，使用属性时文件更小，解析速度更快。很大程度上是因为你不需要基于 XML 结构遍历 DOM，而只是简单地读取属性。

你会考虑使用 XML 吗？鉴于开放 API 如此流行，你经常别无选择。如果只有 XML 格式的数据可用，那么就挽起袖子写代码解析它吧。但是如果有其他格式可用，最好取代它。你在这里看到的标准 XML 的性能数据与更先进的技术相比实在太慢了。如果浏览器支持的话，XPath 可以提高解析速度，但代价是编写并维护三个代码分支（一个针对支持 DOM Level 3 XPath 的浏览器，一个针对 IE 8，还有一个针对其他所有浏览器）。简化版 XML 格式更为有利，但比那些最快的格式依然慢上一个数量级。在高性能 Ajax 中，XML 没有立足之地。

JSON

JSON 是一种使用 JavaScript 对象和数组直接量编写的轻量级且易于解析的数据格式，它由 Douglas Crockford 创立并推广开来[译注3]。下面看一个 JSON 的例子：

[译注2] 关于各浏览器对 XPath 的支持的详细情况，可查阅本书作者 Nicholas C. Zakas 所著的另一本书《JavaScript 高级程序设计》中的相关章节，或查看作者博客中的在线英文文档：*http://www.nczonline.net/blog/tag/xpath/*。

[译注3] 关于 JSON 的更多信息，请参见 *http://www.json.org/json-zh.html*。

```
[
    { "id": 1, "username": "alice", "realname": "Alice Smith",
        "email": "alice@alicesmith.com" },
    { "id": 2, "username": "bob", "realname": "Bob Jones",
        "email": "bob@bobjones.com" },
    { "id": 3, "username": "carol", "realname": "Carol Williams",
        "email": "carol@carolwilliams.com" },
    { "id": 4, "username": "dave", "realname": "Dave Johnson",
        "email": "dave@davejohnson.com" }
]
```

用户表示为一个对象，用户列表表示为一个数组，就像 JavaScript 中其他数组和对象的写法一样。这意味着当它被求值或封装在一个回调函数中时，JSON 数据就是一段可执行的 JavaScript 代码。在 JavaScript 中可以简单地使用`eval()`来解析 JSON 字符串：

```
function parseJSON(responseText) {
    return eval('(' + responseText + ')');
}
```

> **警告**：关于 JSON 和 eval 需要注意的是：在代码中使用 eval 是很危险的，特别是用它执行第三方的 JSON 数据（其中可能包含恶意代码）时。尽可能使用 JSON.parse() 方法解析字符串本身。该方法可以捕获 JSON 中的词法错误，并允许你传入一个函数，用来过滤或转换解析结果。如今此方法已被 Firefox 3.5、IE 8 及 Safari 4 原生支持。大多数 JavaScript 类库包含的 JSON 解析代码会直接调用原生版本，如果没有原生支持的话，会调用一个略微不那么强大的非原生版本来处理。一个非原生版本的参考实现可以从这里找到：*http://json.org/json2.js*。为了保持一致性，eval 将会用在代码示例中。

正如 XML 那样，它可以提炼成一个更为简化的版本。这种情况下，我们可以把属性名缩短（尽管可读性会变差）：

```
[
    { "i": 1, "u": "alice", "r": "Alice Smith", "e": "alice@alicesmith.com" },
    { "i": 2, "u": "bob", "r": "Bob Jones", "e": "bob@bobjones.com" },
    { "i": 3, "u": "carol", "r": "Carol Williams",
        "e": "carol@carolwilliams.com" },
    { "i": 4, "u": "dave", "r": "Dave Johnson", "e": "dave@davejohnson.com" }
]
```

它将相同的数据以更少的结构和更小的文件大小传送给浏览器。更进一步，我们甚至可以去掉属性名。与其他两种格式相比，这种格式可读性更差，也更脆弱，但文件尺寸却小得多：大约只有标准 JSON 的一半。

```
[
    [ 1, "alice", "Alice Smith", "alice@alicesmith.com" ],
    [ 2, "bob", "Bob Jones", "bob@bobjones.com" ],
        [ 3, "carol", "Carol Williams", "carol@carolwilliams.com" ],
        [ 4, "dave", "Dave Johnson", "dave@davejohnson.com" ]
    ]
```

要想成功解析必须保持数据的顺序。虽然是这么说，但你只要在转换格式时按照第一个 JSON 格式保持相同的属性名就好：

```
function parseJSON(responseText) {

    var users = [];
    var usersArray = eval('(' + responseText + ')');

    for (var i = 0, len = usersArray.length; i < len; i++) {
        users[i] = {
            id: usersArray[i][0],
            username: usersArray[i][1],
            realname: usersArray[i][2],
            email: usersArray[i][3]
        };
    }

    return users;
}
```

在这个例子中，我们使用 `eval()` 来转换字符串为原生 JavaScript 数组。二维数组将会转换成一个对象数组。从本质上来讲，就是用一个更复杂的解析函数换取较小的文件尺寸和更快的 `eval()` 时间。下表分别列出了通过 XHR 传送三种 JSON 格式的性能数据。

格式	大小	下载耗时	解析耗时	总耗时
标准 JSON	487 895 字节	527.7 毫秒	26.7 毫秒	554.4 毫秒
简化的 JSON	392 895 字节	498.7 毫秒	29.0 毫秒	527.7 毫秒
数组 JSON	292 895 字节	305.4 毫秒	18.6 毫秒	324.0 毫秒

数组形式的 JSON 在每一项中都取胜了，它文件尺寸最小，平均下载速度最快，平均解析速度也最快。尽管它的解析函数不得不遍历列表中 5 000 个单元，但它的解析速度还是快出了 30%。

JSON-P

事实上，JSON 可以被本地执行会导致几个重要的性能影响。当使用 XHR 时，JSON 数据被当成字符串返回。该字符串紧接着被 eval() 转换成原生对象。然而，在使用动态脚本注入时，JSON 数据被当成另一个 JavaScript 文件并作为原生代码执行。为实现这一点，这些数据必须封装在一个回调函数里。这就是所谓的“JSON 填充(JSON with padding)”或 JSON-P。下面是 JSON-P 格式的用户列表：

```
parseJSON([
    { "id": 1, "username": "alice", "realname": "Alice Smith",
        "email": "alice@alicesmith.com" },
    { "id": 2, "username": "bob", "realname": "Bob Jones",
        "email": "bob@bobjones.com" },
    { "id": 3, "username": "carol", "realname": "Carol Williams",
        "email": "carol@carolwilliams.com" },
    { "id": 4, "username": "dave", "realname": "Dave Johnson",
        "email": "dave@davejohnson.com" }
]);
```

JSON-P 因为回调包装的原因略微增大了文件尺寸，但与其解析性能的提升相比这点增加显得微不足道。由于数据是当作原生的 JavaScript，因此解析速度跟原生 JavaScript 一样快。下面是以 JSON-P 的形式传输三种 JSON 格式的时间对比。

格式	大小	下载耗时	解析耗时	总耗时
Verbose JSON-P	487 913 字节	598.2 毫秒	0.0 毫秒	598.2 毫秒
Simple JSON-P	392 913 字节	454.0 毫秒	3.1 毫秒	457.1 毫秒
Array JSON-P	292 912 字节	316.0 毫秒	3.4 毫秒	319.4 毫秒

文件大小和下载耗时与 XHR 测试基本相同，而解析速度几乎快了 10 倍。标准 JSON-P 的解析耗时为 0，因为它已经是原生格式，所以无须解析。简化版 JSON-P 和数组 JSON-P 同样如此，但是每种都必须通过迭代转换为标准 JSON-P 的格式。

最快的 JSON 格式是使用数组形式的 JSON-P。尽管这只比使用 XHR 的 JSON 稍微快一点，但是这个差异随着列表的增大而增大。如果你的项目需要处理 10000 或 100000 个元素的列表，那么 JSON-P 比 JSON 会好很多。

有一种情形要避免使用 JSON-P（与性能无关），因为 JSON-P 必须是可执行的 JavaScript，它可能被任何人调用并使用动态脚本注入技术插入到任何网站。而另一方面，JSON 在 eval 前是无效的 JavaScript，使用 XHR 时它只是被当作字符串获取。所以不要把任何敏感数据编码在 JSON-P 中，因为你无法确认它是否保持着私有调用状态，即便是带有甚至随机 URL 或 做了 cookie 判断。

应该使用 JSON 吗？

与 XML 相比，JSON 有着许多优点。它有着体积更小的格式，在响应信息中结构所占的比例更小，数据占用得更多。特别是当数据包含数组而不是对象时。JSON 有着极好的通用性，大多数服务端编程语言都提供了编码和解码的类库。它在客户端的解析工作微不足道，

使得你可以花更多的时间来编写代码处理真正有用的数据。对 Web 开发人员来说最重要的是，它是性能表现最好的数据格式，因为它不仅传输尺寸小，而且解析速度快。JSON 是高性能 Ajax 的基础，尤其在使用动态脚本注入时。

HTML

通常你请求的数据需要被转换成 HTML 以显示到页面上。JavaScript 可以比较快地把一个较大的数据结构转换为简单的 HTML，但在服务器处理会快得多。一种可考虑的技术是在服务器上构建好整个 HTML 再传回客户端，JavaScript 可以很方便地通过 `innerHTML` 属性把它插入页面相应的位置。下面是一个 HTML 形式的用户列表的例子：

```
<ul class="users">
    <li class="user" id="1-id002">
        <a href="http://www.site.com/alice/" class="username">alice</a>
        <span class="realname">Alice Smith</span>
        <a href="mailto:alice@alicesmith.com"
            class="email">alice@alicesmith.com</a>
    </li>
    <li class="user" id="2-id002">
        <a href="http://www.site.com/bob/" class="username">bob</a>
        <span class="realname">Bob Jones</span>
        <a href="mailto:bob@bobjones.com" class="email">bob@bobjones.com</a>
    </li>
    <li class="user" id="3-id002">
        <a href="http://www.site.com/carol/" class="username">carol</a>
        <span class="realname">Carol Williams</span>
        <a href="mailto:carol@carolwilliams.com"
            class="email">carol@carolwilliams.com</a>
    </li>
    <li class="user" id="4-id002">
        <a href="http://www.site.com/dave/" class="username">dave</a>
        <span class="realname">Dave Johnson</span>
        <a href="mailto:dave@davejohnson.com"
            class="email">dave@davejohnson.com</a>
    </li>
</ul>
```

这种技术的问题是 HTML 是一种臃肿的数据格式，甚至比 XML 更繁杂。在数据本身的最外层，你可以嵌套 HTML 标签，每个都带有 id、class 和其他属性。HTML 格式可能比实际数据占用更多空间，尽管可以通过尽量少用标签和属性来缓解这一问题。正因为如此，你应当在客户端的瓶颈是 CPU 而不是带宽时才使用此技术。

在一种极端情况下，你有一种格式包含了最少数量的结构，需要在客户端解析数据，例如 JSON。此格式下载到客户端机器的速度非常快，然而，它需要消耗很多的 CPU 时间来转

换成 HTML 格式以显示在页面里。这需要大量的字符串操作，而字符串操作也是 JavaScript 中最慢的部分。

在另一种极端情况下，你得到服务器创建好的 HTML。这种格式在线传输的数据量可能比较大，下载花的时间也会更长，但一旦下载完成，只需一次操作就能把它显示在页面上：

```
document.getElementById('data-container').innerHTML = req.responseText;
```

下表显示了使用 HTML 格式的用户列表的性能数据。请记住此格式与其他格式的区别是："解析"在这种情况下是指把 HTML 插入 DOM 的动作。此外，HTML 不能像原生 JavaScript 数组那样方便而快速地进行迭代操作。

格式	大小	下载耗时	解析耗时	总耗时
HTML	1 063 416 字节	273.1 毫秒	121.4 毫秒	394.5 毫秒

正如你所看到的，HTML 传输数据量明显偏大，还需要较长的时间来解析。这是因为把 HTML 插入 DOM 的单一操作看似简单，尽管它只有一行代码，却需要大量时间把更多数据载入页面。与其他数据相比，这些性能数据和其他格式的数据相比有些许偏差，因为最终的结果不是数据数组，而是显示在页面上的 HTML 元素。无论如何，它仍然说明了关于 HTML 的一个事实：作为一种数据格式，它既缓慢，又臃肿。

自定义格式

Custom Formatting

理想的数据格式应该只包含必要的结构，以便你可以分解出每一个独立的字段。你可以很容易地定义一种这样的格式，只须简单地把数据用分隔符连接起来：

```
Jacob;Michael;Joshua;Matthew;Andrew;Christopher;Joseph;Daniel;Nicholas;
Ethan;William;Anthony;Ryan;David;Tyler;John
```

这些分隔符本质上创建了一个数据数组，类似于一个用逗号分隔的列表。通过使用不同的分隔符，你可以创建多维数组。下面是一个用字符串分隔的自定义格式的用户列表：

```
1:alice:Alice Smith:alice@alicesmith.com;
2:bob:Bob Jones:bob@bobjones.com;
3:carol:Carol Williams:carol@carolwilliams.com;
4:dave:Dave Johnson:dave@davejohnson.com
```

这种格式非常简洁，"数据/结构"的比例相当高（比其他任何格式都高出很多，不包括纯文本）。自定义格式可以很快速地下载，且易于解析，只需简单地调用字符串的 `split()`并传入分隔符作为参数即可。更为复杂的具有多个分隔符的自定义格式需要在循环中分离所有数

据（但请记住，JavaScript 中执行这些循环是非常快的）。`split()`是最快的字符串操作方式之一，通常它能在数毫秒内处理包含 10 000+个元素的“分隔符分隔”列表。下面是一个解析以上格式的例子：

```
function parseCustomFormat(responseText) {

    var users = [];
    var usersEncoded = responseText.split(';');
    var userArray;

    for (var i = 0, len = usersEncoded.length; i < len; i++) {

        userArray = usersEncoded[i].split(':');

        users[i] = {
            id: userArray[0],
            username: userArray[1],
            realname: userArray[2],
            email: userArray[3]
        };
    }

    return users;
}
```

当你创建自定义格式时，最重要的决定之一就是采用哪种分隔符。理想情况下，它应当是一个单字符，而且不应该存在于你的数据中。ASCII 字符表的前几个字符在大多数服务端语言中能够正常工作而且容易书写。例如，下面演示了如何在 PHP 中使用 ASCII 字符：

```
function build_format_custom($users) {

    $row_delimiter = chr(1);  // JavaScript 中的 \u0001
    $field_delimiter = chr(2); // JavaScript 中的 \u0002

    $output = array();
    foreach ($users as $user) {
        $fields = array($user['id'], $user['username'], $user['realname'],
$user['email']);
        $output[] = implode($field_delimiter, $fields);
    }

    return implode($row_delimiter, $output);
}
```

这些控制字符在 JavaScript 中使用 Unicode 表示法（例如`\u0001`）。`split()`函数可以使用字符串或正则表达式作参数。如果你预期在数据中存在空白字段，那么就使用字符串作为分隔符；如果分隔符是一个正则表达式，IE 中的 `split()` 将会忽略紧挨着的两个分隔符中的第二个分隔符[译注4]。这两种参数类型在其他浏览器上是等价的。

[译注4] 本书第 5 章的作者“Steven Levithan”有一篇文章专门分析了 split 函数接受正则表达式参数时在不同浏览器下的差异，请参见 *http://blog.stevenlevithan.com/archives/cross-browser-split*，他还提供了一个详细的 demo 页面测试你当前使用的浏览器对 split 函数接受正则表达式参数的情形是否按照 ECMAScript 规范正确处理的：*http://stevenlevithan.com/demo/split.cfm*。

```
// 正则表达式作为分隔符
var rows = req.responseText.split(/\u0001/);

// 字符串作为分隔符（更为保险）
var rows = req.responseText.split("\u0001");
```

下面是以字符分隔的自定义格式的性能数据，分别使用 XHR 和动态脚本注入：

格式	大小	下载耗时	解析耗时	总耗时
Custom Format (XHR)	222 892 字节	63.1 毫秒	14.5 毫秒	77.6 毫秒
Custom Format (script insertion)	222 912 字节	66.3 毫秒	11.7 毫秒	78.0 毫秒

XHR 和动态脚本注入都可以使用这种格式。两种情况下都要解析字符串，因此没有实质上的性能差异。对于非常大的数据集，它是最快的格式，甚至在解析速度和平均加载时间上都能击败本地执行的 JSON。当你需要在很短的时间内向客户端传送大量数据时可以考虑使用这种格式。

数据格式总结

Data Format Conclusions

通常来说数据格式越轻量级越好，JSON 和字符分隔的自定义格式是最好的。如果数据集很大并且对解析时间有要求，那么就从如下两种格式中做出选择：

- JSON-P 数据，使用动态脚本注入获取。它把数据当作可执行 JavaScript 而不是字符串，解析速度极快。它能跨域使用，但涉及敏感数据时不应该使用它。
- 字符分隔的自定义格式，使用 XHR 或动态脚本注入获取，用 `split()`解析。这项技术解析大数据集比 JSON-P 略快，而且通常文件尺寸更小。

下表和图 7-1 再次显示了所有性能数据（由慢到快排序），因此你可以同时比较每种格式。其中不包括 HTML，因为它与其他格式无法直接比较。

格式	大小	下载耗时	解析耗时	总耗时
标准 XML	582 960 字节	999.4 毫秒	343.1 毫秒	1342.5 毫秒
标准 JSON-P	487 913 字节	598.2 毫秒	0.0 毫秒	598.2 毫秒
简化的 XML	437 960 字节	475.1 毫秒	83.1 毫秒	558.2 毫秒
标准的 JSON	487 895 字节	527.7 毫秒	26.7 毫秒	554.4 毫秒
简化的 JSON	392 895 字节	498.7 毫秒	29.0 毫秒	527.7 毫秒
简化的 JSON-P	392 913 字节	454.0 毫秒	3.1 毫秒	457.1 毫秒

续表

格式	大小	下载耗时	解析耗时	总耗时
数组 JSON	292 895 字节	305.4 毫秒	18.6 毫秒	324.0 毫秒
数组 JSON-P	292 912 字节	316.0 毫秒	3.4 毫秒	319.4 毫秒
自定义格式(脚本注入)	222 912 字节	66.3 毫秒	11.7 毫秒	78.0 毫秒
自定义格式（XHR）	222 892 字节	63.1 毫秒	14.5 毫秒	77.6 毫秒

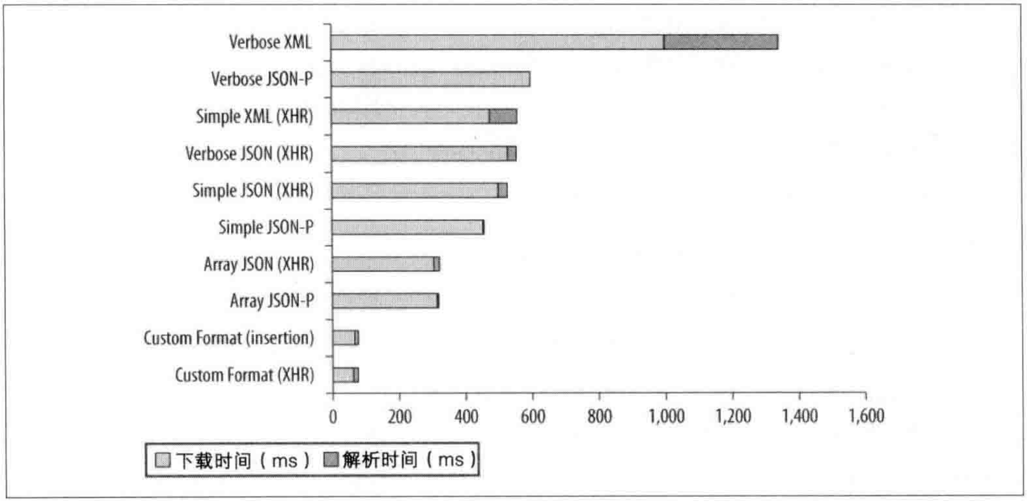


图 7-1　各种数据格式下载和解析时间的对比

请注意，这些数据只是在一个浏览器上进行一次测试获得的。此结果可用作大概的性能指标，并非确切数字。你可以自己运行这些测试：*http://techfoolery.com/formats/*。

Ajax 性能指南

Ajax Performance Guidelines

一旦你选择了最适合的数据传输技术和数据格式，那么你就可以开始考虑其他优化技术了。这些技术要根据具体情况来使用，因此在考虑使用它们之前你要确认它们适用于你的应用程序。

缓存数据

Cache Data

最快的 Ajax 请求就是没有请求。有两种主要的方法可避免发送不必要的请求：

- 在服务端，设置 HTTP 头信息以确保你的响应会被浏览器缓存。
- 在客户端，把获取到的信息存储到本地，从而避免再次请求。

第一种技术使用最简单而且好维护，而第二种则给你最大的控制权。

设置 HTTP 头信息

如果你希望 Ajax 响应能够被浏览器缓存，那么你必须使用 GET 方式发出请求。但这还不够，你还必须在响应中发送正确的 HTTP 头信息。`Expires` 头信息会告诉浏览器应该缓存响应多久。它的值是一个日期，过期之后，对该 URL 的任何请求都不再从缓存中获取，而是会重新访问服务器。一个 `Expires` 头信息格式如下：

```
Expires: Mon, 28 Jul 2014 23:30:00 GMT
```

这种特殊的 `Expires` 头信息告诉浏览器缓存此响应到 2014 年 7 月。这种 `Expires` 头信息被称为“遥远的未来”，它对那些从不改变的内容非常有用，比如图片和其他静态数据集。

`Expires` 头信息中的日期是 GMT 日期。在 PHP 中可使用如下代码设置：

```
$lifetime = 7 * 24 * 60 * 60; // 7 天，按秒计算
header('Expires: ' . gmdate('D, d M Y H:i:s', time() + $lifetime) . ' GMT');
```

以上代码会告诉浏览器缓存此文件 7 天。如果要设置一个遥远未来的 `Expires` 头信息，就把生命周期设置得更长。以下代码告诉浏览器缓存此文件 10 年：

```
$lifetime = 10 * 365 * 24 * 60 * 60; // 10 年，按秒计算
header('Expires: ' . gmdate('D, d M Y H:i:s', time() + $lifetime) . ' GMT');
```

设置 `Expires` 头信息是确保浏览器缓存 Ajax 响应最简单的方法。你无须改变客户端的任何代码，可以继续像平时那样发送 Ajax 请求，但你知道浏览器只会当文件不存在缓存中时才会向服务器发送请求。它在服务端也很容易实现，所有的语言都允许你通过某种方法设置头信息。这是保证你的数据被缓存的最简单的方法。

本地数据存储

除了依赖浏览器处理缓存外，你还可以用更手工的方式来实现，即直接把从服务器接收到的数据储存起来。你可以把响应文本保存到一个对象中，以 URL 为键值作为索引。下面的例子对 XHR 作了封装，它首先会检查 URL 是否获取过：

```
var localCache = {};

function xhrRequest(url, callback) {

    // 检查此 URL 的本地缓存

    if (localCache[url]) {
        callback.success(localCache[url]);
        return;
    }

    // 此 URL 对应的缓存没找到，则发送请求

    var req = createXhrObject();
    req.onerror = function() {
        callback.error();
    };

    req.onreadystatechange = function() {
        if (req.readyState == 4) {

            if (req.responseText === '' || req.status == '404') {
                callback.error();
                return;
            }

            // 存储响应文本到本地缓存

            localCache[url] = req.responseText;
            callback.success(req.responseText);
        }
    };

    req.open("GET", url, true);
    req.send(null);
}
```

总的来说，设置一个 `Expires` 头信息是更好的方案。它实现起来比较容易，而且其缓存内容能跨页面和跨会话（session）。而手工管理的缓存在你希望编程废止缓存内容并获取更新数据的时候会很有用。设想这样一种情况，你每次请求都会用到缓存数据，但用户执行了某些动作可能导致一个或多个已缓存的响应失效。这种情况下从缓存中删除那些响应十分简单：

```
delete localCache['/user/friendlist/'];
delete localCache['/user/contactlist/'];
```

本地缓存也可以很好地工作在移动设备上。此类设备上的大多数浏览器的缓存都很小或者没有缓存，手工管理的缓存成为避免不必要请求的最佳选择。

了解 Ajax 类库的局限

Know the Limitations of Your Ajax Library

所有的 JavaScript 类库都允许你访问一个 Ajax 对象，它屏蔽了浏览器之间的差异，给你一个统一的接口。大多数情况下这都是件好事情，使得你可以专注于项目而不是 XHR 在那些古怪的浏览器上的工作细节。然而，为了给你一个统一的接口，这些类库必须简化接口，因为并不是所有浏览器都实现了所需的每一项功能。这使得你不能访问 `XMLHttpRequst` 的完整功能。

本章讨论到的一些技术只能通过直接访问 XHR 对象来实现。其中最值得注意的是在 multipart XHR 中使用了流功能。通过监听 `readyState` 为 3 的状态，我们可以在一个较大的响应还没有完全接受之前把它分段处理。这使得我们可以实时处理响应片段，这也是 MXHR 能大幅提升性能的主要原因。然而大多数 JavaScript 类库不允许你直接访问 `readystatechange` 事件。这意味着你必须等待完整的响应接收完毕（可能是一个相当长的时间）然后才能开始使用它。

直接使用 `XMLHttpRequest` 对象并非像它看上去的那样可怕。除了少数个别情况，大部分最近的主流浏览器都以相同的方式支持 `XMLHttpRequest` 对象，也都提供了对不同的 `readyStates` 状态的访问。你只需要很少几行代码就可以兼容老版本的 IE。下面的例子中返回一个 XHR 对象，你可以直接使用它（以下代码来自于 YUI 2 中的 Connection Manager 对象的源码，稍有调整）：

```
function createXhrObject() {

    var msxml_progid = [
        'MSXML2.XMLHTTP.6.0',
        'MSXML3.XMLHTTP',
        'Microsoft.XMLHTTP', // 不支持 readyState 3
        'MSXML2.XMLHTTP.3.0', // 不支持 readyState 3
    ];

    var req;
    try {
        req = new XMLHttpRequest(); // 先尝试标准方法
    }
    catch(e) {
        for (var i = 0, len = msxml_progid.length; i < len; ++i) {
            try {
                req = new ActiveXObject(msxml_progid[i]);
                break;
            }
            catch(e2) {  }
        }
    }
    finally {
        return req;
    }
}
```

这段代码首先会尝试支持 `readyState` 3 的 `XMLHttp` 控件版本，然后降级到那些不支持此状态的版本。

直接操作 XHR 对象减少了函数开销，进一步提升了性能。需要注意的是，如果放弃使用 Ajax 类库，那么你可能在一些古怪的浏览器上遇到一些问题。

小结

Summary

高性能的 Ajax 包括以下方面：了解你项目的具体需求，选择正确的数据格式和与之匹配的传输技术。

作为数据格式，纯文本和 HTML 只适用于特定场合，但它们可以节省客户端的 CPU 周期。XML 被广泛应用而且支持良好，但是它十分笨重且解析缓慢。JSON 是轻量级的，解析速度快（被视为原生代码而不是字符串），通用性与 XML 相当。字符分隔的自定义格式十分轻量，在解析大量数据集时非常快，但需要编写额外的服务端构造程序，并在客户端解析。

当从页面当前所处的域下请求数据时，XHR 提供了最完善的控制和灵活性，尽管它会把接收到的所有数据当成一个字符串，且这有可能降低解析速度。另一方面，动态脚本注入允许跨域请求和本地执行 JavaScript 和 JSON 但是它的接口不那么安全，而且还不能读取头信息或响应代码。Multipart XHR 可以用来减少请求数，并处理一个响应中的各种文件类型，但是它不能缓存接收到的响应。当需要发送数据时，图片信标是一种简单而有效的方法。XHR 还可以用 POST 方法发送大量数据。

除了这些格式和传输技术，还有一些准则有助于加速你的 Ajax：

- 减少请求数，可通过合并 JavaScript 和 CSS 文件，或使用 MXHR。
- 缩短页面的加载时间，页面主要内容加载完成后，用 Ajax 获取那些次要的文件。
- 确保你的代码错误不会输出给用户，并在服务端处理错误。
- 知道何时使用成熟的 Ajax 类库，以及何时编写自己的底层 Ajax 代码。

Ajax 为提升你的网站的潜在性能提供了广阔的空间，因为许多网站大量使用异步请求，而且它还提供了一些与它无关的问题的解决方案，比如有过多的资源需要加载。对 XHR 创造性地使用是一个反应迟钝且平淡无奇的页面与响应快速且高效的页面的区别所在；是一个用户痛恨使用的站点与用户迷恋的站点的区别所在。

第 8 章

编程实践

Programming Practices

每种编程语言都有它的“痛点”，并且随着时间的推移，它存在的低效模式也不断发展。其原因在于，越来越多的人们开始转向这门语言，并不断扩充它的边界。自 2005 年以来的“Ajax”的热潮出现后，Web 开发者对 JavaScript 和浏览器的推动作用远远超过以往。结果是出现了一些十分特别的模式，有精华也有糟粕。这些模式的出现是由 Web 中 JavaScript 的特性决定的。

避免双重求值（Double Evaluation）

Avoid Double Evaluation

JavaScript 像其他很多脚本语言一样，允许你在程序中提取一个包含代码的字符串，然后动态执行它。有四种标准方法可以实现：`eval()`、`Function()` 构造函数、`setTimeout()` 和 `setInterval()`。其中每个方法都允许你传入一个 JavaScript 代码字符串并执行它。来看几个例子：

```
var num1 = 5,
    num2 = 6,

    // eval()执行代码字符串
    result = eval("num1  +  num2"),

    //Function() 执行代码字符串
    sum = new Function("arg1", "arg2", "return arg1 + arg2");

//setTimeout() 执行代码字符串
setTimeout("sum = num1 + num2", 100);

//setInterval() 执行代码字符串
setInterval("sum = num1 + num2", 100);
```

当你在 JavaScript 代码中执行另一段 JavaScript 代码时，都会导致双重求值的性能消耗。此代码首先会以正常的方式求值，然后在执行过程中对包含于字符串中的代码发起另一个求

值运算。双重求值是一项代价昂贵的操作，它比直接包含的代码执行速度慢许多。

作为一个比较点，各种浏览器访问一个数组条目的时间各有不同，但如果使用 `eval()`来访问数组条目的结果则大有区别，如例所示：

```
//较快
var item = array[0];

//较慢
var item = eval("array[0]");
```

如果使用 `eval()`替代原生代码读取 10 000 个数组条目，在不同浏览器上的差异非常大。表 8-1 显示了此操作所用的时间。

表 8-1　用原生代码与 eval()分别读取 10 000 个数组项的速度对比

浏览器类别	原生代码（毫秒）	eval()（毫秒）
Firefox 3	10.57	822.62
Firefox 3.5	0.72	141.54
Chrome 1	5.7	106.41
Chrome 2	5.17	54.55
Internet Explorer 7	31.25	5086.13
Internet Explorer 8	40.06	420.55
Opera 9.64	2.01	402.82
Opera 10 Beta	10.52	315.16
Safari 3.2	30.37	360.6
Safari 4	22.16	54.47

访问数组条目所耗的时间存在巨大差异是因为，每次调用 `eval()` 时都要创建一个新的解释器/编译器实例。同样的过程也发生在使用 `Function()`、`setTimeout()` 和 `setInterval()` 时，这必然使得代码执行的速度变慢。

大多数时候，没必要使用 `eval()`和 `Function()`，因此最好避免使用它们。至于另外两个函数：`setTimeout()`和 `setInterval()`，建议传入函数而不是字符串来作为第一个参数。比如：

```
setTimeout(function(){
    sum = num1 + num2;
}, 100);

setInterval(function(){
    sum = num1 + num2;
}, 100);
```

避免双重求值是实现 JavaScript 运行期性能最优化的关键所在。

提示：优化后的 JavaScript 引擎通常会缓存住那些使用了 `eval()` 且重复运行的代码。如果你在 Safari 4 和所有版本 Chrome 中对同一段代码字符串反复求值，你会看到显著的性能提升。

使用 Object/Array 直接量

Use Object/Array Literals

在 JavaScript 中创建对象和数组的方法有多种，但使用对象和数组直接量是最快的方式。如果不使用直接量，那么典型的对象创建和赋值的写法如下：

```
//创建一个对象
var  myObject = new  Object();
myObject.name = "Nicholas";
myObject.count = 50;
myObject.flag = true;
myObject.pointer = null;

//创建一个数组
var  myArray = new Array();
myArray[0] = "Nicholas";
myArray[1] = 50;
myArray[2] = true;
myArray[3] = null;
```

尽管从技术角度而言，以上代码并没有问题，但直接量会运行得更快。还有一个额外的好处是，直接量还有助于节省代码量，以减小整个文件的尺寸。上面的代码可以用直接量重写如下：

```
//创建一个对象
var myObject = {
    name: "Nicholas",
    count: 50,
    flag: true,
    pointer: null
};

//创建一个数组
var myArray = ["Nicholas", 50, true, null];
```

两段代码的执行结果是相同的，但后者在浏览器中运行得更快（Firefox 3.5 中似乎并无差异）。对象属性和数组项的数量越多，使用直接量的好处就越明显。

避免重复工作
Don't Repeat Work

在计算机科学领域中最主要的性能优化技术之一就是“避免无谓的工作（work avoidance）”。避免无谓的工作的概念有两重意思：别做无关紧要的工作，别重复做已经完成的工作。第一部分通常很容易在代码重构时发现。而第二部分往往难以界定，因为工作可能会在各种场景下因为各种理由而被重复。

也许最常见的重复工作就是浏览器探测。基于浏览器的功能作分支判断导致产生大量的代码。考虑一个添加和移除事件处理器的例子。典型的跨浏览器代码写法如下：

```
function addHandler(target, eventType, handler){
    if (target.addEventListener){  //DOM2 Events
        target.addEventListener(eventType, handler, false);
    } else {   //IE
        target.attachEvent("on" + eventType, handler);
    }
}

function removeHandler(target, eventType, handler){
    if (target.removeEventListener){  //DOM2 Events
        target.removeEventListener(eventType, handler, false);
    } else { //IE
        target.detachEvent("on" + eventType, handler);
    }
}
```

此代码通过测试 `addEventListener()`和 `removeEventListener()`来检查 DOM Level2 事件是否被支持。除 IE 外所有主流浏览器支持 DOM Level 2 事件。如果此方法在 `target` 中不存在，则假定当前浏览器是 IE，并使用 IE 特有的方法。

乍一看这些函数似乎已经过充分优化。隐藏的性能问题在于每次函数调用时都做了重复工作。因为每次的检查过程都相同：看看指定方法是否存在。如果你假定 `target` 唯一的值就是 DOM 对象，而且用户不可能在页面加载完后奇迹般地改变浏览器，那么这次检查就是重复的。如果在第一次调用 `addHandler()`时就确定 `addEventListener()`是存在的，那么随后每次调用时它应该也都存在。每次调用一个函数都重复相同的工作是一种浪费。有几种方法可以避免它。

延迟加载
Lazy Loading

第一种消除函数中的重复工作的方法是延迟加载（lazy loading）。延迟加载意味着在信息被使用前不会做任何操作。比如前面的例子，在函数被调用前，没有必要判断该用哪个方法

去绑定或取消绑定事件处理器。采用延迟加载的函数版本如下：

```
function addHandler(target, eventType, handler){

    //复写现有函数
    if (target.addEventListener){ //DOM2 Events
        addHandler = function(target, eventType, handler){
            target.addEventListener(eventType, handler, false);
        };
    } else { //IE
        addHandler = function(target, eventType, handler){
            target.attachEvent("on" + eventType, handler);
        };
    }

    //调用新函数
    addHandler(target, eventType, handler);
}

function removeHandler(target, eventType, handler){

    //复写现有函数
    if (target.removeEventListener){    //DOM2 Events
        removeHandler = function(target, eventType, handler){
            target.addEventListener(eventType, handler, false);
        };
    } else { //IE
        removeHandler = function(target, eventType, handler){
            target.detachEvent("on" + eventType, handler);
        };
    }

    //调用新函数
   removeHandler(target, eventType, handler);
}
```

这两个函数实现了延迟加载模式。这两个方法在第一次被调用时，会先检查并决定使用哪种方法去绑定或取消绑定事件处理器。然后原始函数被包含正确操作的新函数覆盖。最后一步调用新的函数，并传入原始参数。随后每次调用 `addHandler()`或 `removeHandler()`都不会再做检测，因为检测代码已经被新的函数覆盖。

调用延迟加载函数时，第一次总会消耗较长的时间，因为它必须运行检测接着再调用另一个函数完成任务。但随后调用相同的函数会更快，因为不需要再执行检测逻辑。当一个函数在页面中不会立刻调用时，延迟加载是最好的选择。

条件预加载

Conditional Advance Loading

除了延迟加载函数之外的另一种选择是：条件预加载（conditional advanceloading），它会在脚本加载期间提前检测，而不会等到函数被调用。检测的操作依然只有一次，只是它在过程中来得更早。例如：

```
var addHandler = document.body.addEventListener ?
                 function(target, eventType, handler){
                     target.addEventListener(eventType, handler, false);
                 }:
                 function(target, eventType, handler){
                     target.attachEvent("on" + eventType, handler);
                 };

var removeHandler = document.body.removeEventListener ?
                 function(target, eventType, handler){
                     target.removeEventListener(eventType, handler, false);
                 }:
                 function(target, eventType, handler){
                     target.detachEvent("on" + eventType, handler);
                 };
```

这个例子先检查`addEventListener()`和`removeEventListener()`是否存在，然后根据结果指定选择最佳的函数。如果它们存在的话，三元运算符会返回 DOM Level 2 函数，否则返回 IE 特有的函数。这样做的结果是所有对`allHandler()`和`removeHandler()`的调用都十分快，因为检测提前发生了。

条件预加载确保所有函数调用消耗的时间相同。其代价是需要在脚本加载时就检测，而不是加载后。预加载适用于一个函数马上就要被用到，并且在整个页面的生命周期中频繁出现的场合。

使用速度快的部分

Use the Fast Parts

尽管 JavaScript 经常被指责运行缓慢，但这门语言的某些部分运行却快得让人难以置信。这并不足为奇，因为 JavaScript 引擎是由低级语言构建的而且经过编译。虽然 JavaScript 运行速度慢很容易被归咎于引擎，然而引擎通常是处理过程中最快的部分，运行速度慢的实际上是你的代码。引擎的某些部分比其他部分快很多，因为它们允许你绕过那些慢的部分。

位操作

Bitwise Operators

位操作是 JavaScript 中最容易被误解的方面之一。普遍认为，开发者们不知道如何使用这些操作符，而且经常被误用在布尔表达式中。于是就导致 JavaScript 开发中很少使用位操作符，尽管它们具有优势。

JavaScript 中的数字都依照 IEEE-754 标准以 64 位格式存储。在位操作中，数字被转换为有符号 32 位格式。每次运算符会直接操作该 32 位数以得到结果。尽管需要转换，但这个过程与 JavaScript 其他数学运算和布尔操作相比要快很多。

如果你不太熟悉数字的二进制表示法，用 JavaScript 中的 `toString()`方法并传入参数 2 能很容易地把数字转换为字符串形式的二进制表达式。例如：

```
var num1 = 25,
    num2 = 3;

alert(num1.toString(2)); //"11001"
alert(num2.toString(2)); //"11"
```

请记住，此表达式忽略了数字高位的零。JavaScript 中有四种位逻辑操作符：

Bitwise AND **按位与**

两个操作数的对应位都是 1 时，则在该位返回 1。

Bitwise OR **按位或**

两个操作数的对应位只要一个为 1 时，则在该位返回 1。

Bitwise XOR **按位异或**

两个操作数的对应位只有一个为 1，则在该位返回 1。

Bitwise NOT **按位取反**

遇 0 则返回 1，反之亦然。

这些操作符的用法如下：

```
//bitwise AND //按位与
var result1 = 25 & 3;          //1
alert(result1.toString(2)); //"1"

//bitwise OR //按位或
var result2 = 25 | 3;          //27
alert(result2.toString(2)); //"11011"

//bitwise XOR //按位异或
var result3 = 25 ^ 3;          //26
alert(result3.toString(2)); //"11010"

//bitwise NOT //按位取反
var result4 = ~25;             //-26
alert(result4.toString(2)); //"-11010"
```

有好几种方法来利用位操作符提升 JavaScript 的速度。首先是使用位运算代替纯数学操作。比如通常采用对 2 取模运算实现表格行颜色交替，例如：

```
for (var i=0, len=rows.length; i < len; i++){

        if (i % 2) {
            className = "even";
        } else {
            className = "odd";
        }

        // 增加 class
    }
```

对 2 的取模计算，需要用这个数除以 2 然后查看余数。如果你看到 32 位数字的二进制底层表示，你会发现偶数的最低位是 0，奇数的最低位是 1。这可以简单地通过让给定数字与数字 1 进行按位与运算判断出来。当此数为偶数，那么它和 1 进行按位与运算的结果是 0；如果此数为奇数，那么它和 1 进行按位与运算的结果就是 1。也就是说上面的代码可以重写如下：

```
for (var i=0, len=rows.length; i < len; i++){
    if (i & 1) {
        className = "odd";
    } else {
        className = "even";
    }

    // 增加 class
}
```

虽然代码改动不大，但按位与版本比原始版本快了 50%（取决于浏览器）。

第二种使用位运算的技术称作“位掩码”。位掩码是计算机科学中一种常用的技术，用于处理同时存在多个布尔选项的情形。其思路即使用单个数字的每一位来判定是否选项成立，从而有效地把数字转换为由布尔值标记组成的数组。掩码中的每个选项的值都等于 2 的幂。例如：

```
var OPTION_A = 1;
var OPTION_B = 2;
var OPTION_C = 4;
var OPTION_D = 8;
var OPTION_E = 16;
```

通过定义这些选项，你可以用按位或运算创建一个数字来包含多个设置选项：

```
var options = OPTION_A | OPTION_C | OPTION_D;
```

接下来你可以通过按位与操作来判断一个给定的选项是否可用。如果该选项未设置则运算结果为 0，如果已设置则结果为 1：

```
//选项 A 是否在列表中?
if (options & OPTION_A){
    // 代码处理
}

//选项 B 是否在列表中?
if (options & OPTION_B){
    // 代码处理
}
```

像这样的位掩码运算速度非常快，原因正如前面提到的，计算操作发生在系统底层。如果有许多选项保存在一起并频繁检查，位掩码有助于提高整体性能。

提示：JavaScript 也支持按位左移（<<），按位右移（>>），和无符号右移等位运算符。

原生方法

Native Methods

无论你的 JavaScript 代码如何优化，都永远不会比 JavaScript 引擎提供的原生方法更快。道理很简单：JavaScript 的原生部分在你写代码前已经存在浏览器中了，并且都是用低级语言写的，诸如 C++。这意味着这些方法会被编译成机器码，成为浏览器的一部分，所以不会像你写的 JavaScript 代码那样受到各种限制。

经验不足的 JavaScript 开发者经常犯的一个错误是在代码中进行复杂的数学运算，而没有使用内置的 `Math` 对象中那些性能更好的版本。`Math` 对象中那些特别设计的属性和方法是为了让数学运算变得更容易。这里是一些常见的数学常量：

常量	值
`Math.E`	E 的值，自然对数的底
`Math.LN10`	10 的自然对数
`Math.LN2`	2 的自然对数
`Math.LOG2E`	以 2 为底的 E 的对数
`Math.LOG10E`	以 10 为底的 E 的对数
`Math.PI`	π 常量值
`Math.SQRT1_2`	1/2 的平方根
`Math.SQRT2`	2 的平方根

这里的每个数值都是预先计算好的，因此你不需要自己来计算它们。这里还有一些处理数学运算的方法：

方法	含义
Math.abs(*num*)	返回 *num* 的绝对值
Math.exp(*num*)	返回 E 的指数
Math.log(*num*)	返回 *num* 的自然对数
Math.pow(*num*,*power*)	返回 *num* 的 *power* 次幂
Math.sqrt(*num*)	返回 *num* 的平方根
Math.acos(*x*)	返回 *x* 的反余弦值
Math.asin(*x*)	返回 *x* 的反正弦值
Math.atan(*x*)	返回 *x* 的反正切值
Math.atan2(*y*,*x*)	返回从 X 轴到(*y*,*x*)点的角度
Math.cos(*x*)	返回 *x* 的余弦值
Math.sin(*x*)	返回 *x* 的正弦值
Math.tan(*x*)	返回 *x* 的正切值

使用这些方法比使用同样功能的 JavaScript 代码更快。当你需要运行复杂数学运算时，应先查看 Math 对象。

另一个例子是选择器 API，它允许使用 CSS 选择器来查找 DOM 节点。CSS 查询被 JavaScript 原生支持并被 jQuery 发扬光大。jQuery 引擎被广泛认为是最快的 CSS 查询引擎，但是它仍然比原生方法慢。原生的 `querySelector()` 和 `querySelectorAll()` 方法完成任务平均所需时间是基于 JavaScript 的 CSS 查询的 10%[注1]。大部分 JavaScript 类库已经开始转向尽量使用原生方法以提升它们的整体性能。

当原生方法可用时，尽量使用它们。特别是数学运算和 DOM 操作。用编译后的代码做越多的事情，你的代码就会越快。According to the SlickSpeed test suite at.

提示：Chrome 实际上实现了相当数量的 JavaScript 原生方法。因为 Chrome 对原生方法和你的代码都使用即时 JavaScript 编译器，所以它们之间的性能差别不大。

[注1] 统计数据来自测试工具 SlickSpeed：*http://www2.webkit.org/perf/slickspeed/*。

小结

Summary

JavaScript 提出了一些独一无二的性能挑战，这与你组织代码的方式有关。随着 Web 应用变得越来越高级，包含的 JavaScript 代码也越来越多，各种模式和反模式[译注1]也逐渐出现。为了编写更高效的代码，请牢记以下编程实践：

- 通过避免使用 `eval()`和 `Function()`构造器来避免双重求值带来的性能消耗。同样的，给 `setTimeout()`和 `setInterval()`传递函数而不是字符串作为参数。
- 尽量使用直接量创建对象和数组。直接量的创建和初始化都比非直接量形式要快。
- 避免做重复的工作。当需要检测浏览器时，可使用延迟加载或条件预加载。
- 在进行数学计算时，考虑使用直接操作数字的二进制形式的位运算。
- JavaScript 的原生方法总会比你写的任何代码都要快。尽量使用原生方法。

当本书涵盖了大量的优化技术和方法，当把这些方案应用在那些被频繁运行的代码上时，你将会看到显著的性能提升。

[译注1] 参见维基百科：*http://zh.wikipedia.org/zh-cn/*反模式。

第 9 章

构建并部署高性能 JavaScript 应用

Building and Deploying High-Performance JavaScript Applications

Julien Lecomte

根据 Yahoo! 特别性能小组在 2007 的研究，40%~60% 的 Yahoo! 用户没有使用缓存的经验，大概 20% 的页面 PV 是在无缓存的状态下被访问的（*http://yuiblog.com/blog/2007/01/04/performance-research-part-2/*）。另外，由 Yahoo! 搜索小组发现，并由 Google 的 Steve Souders 证实的一项最新研究表明，大约 15% 的美国大型网站提供的内容未经压缩。

这些事实强调有必要确保那些基于 JavaScript 的 Web 应用尽可能高效地传输。虽然部分工作在设计和开发阶段已经完成，但构建和部署过程也很重要但经常被忽视。如果在这个环节不够小心，你的应用的性能将受到影响，无论你怎样努力都无法使它更快。

本章的目的是让你了解如何有效地组织并部署基于 JavaScript 的 Web 应用的一些必要的知识。许多概念将用 Apache Ant 说明，它是一个基于 Java 的构建工具，并很快成为构建 Web 应用的工业标准。在本章末尾，给出了一个用 PHP5 编写的可灵活定制的构建工具示例。

Apache Ant [译注1]

Apache Ant（*http://ant.apache.org/*）是一个软件构建自动化工具。它类似于 make，但是用 Java 实现并用 XML 描述构建过程，而 make 则使用自有的 Makefile 格式。Ant 是 Apache Software Foundation（*http://www.apache.org/licenses/*）的项目之一。

[译注1] 在本书首次出版至今的 5 年里，前端自动化构建工具层出不穷，相比经典的 Apache Ant，Glup 则是一个更为主流的选择，详见 http://gulpjs.com。

相比 `make` 及其他构建工具，Ant 的优势在于它的可移植性。Ant 可以用在许多不同平台上，而且 Ant 构建文件的格式与平台无关。

Ant 构建文件是 XML 格式，默认名字为 build.xml。每个构建文件包含唯一一个 project[译注2]和至少一个 target[译注3]。一个 Ant target 可依赖其他 target。

Target 包含任务元素，即自动执行的动作。Ant 内置了大量任务，如果需要还可以添加可选任务[译注4]。此外，Ant 构建文件里用到的自定义任务可以使用 Java 开发。

一个 project 可以设置一系列属性或变量。一个属性具有一个名字和一个值。它可由构建文件中的 `property` 任务定义，也能在 Ant 外部定义。引用属性的方法是：把属性名放在“`${`和`}`”中间。

以下是一个构建文件的例子。运行默认 target（`dist`）编译源目录中的 Java 代码并打包成一个 JAR 文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MyProject" default="dist" basedir=".">

  <!-设置本次构建的全局属性 ->
  <property name="src"   location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
    <!-- 创建时间戳 -->
    <tstamp/>
    <!-- 创建用于编译的 build 目录结构 -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init" description="compile the source">
    <!-- 把 ${src} 中的 Java 代码编译成 ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile" description="generate the distribution">
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>
    <!-- 把 ${build} 目录的文件打包成 MyProject-${DSTAMP}.jar 文件 -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean" description="clean up">
    <!-- 删除 ${build} 和 ${dist} 目录树 -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>

</project>
```

译注2: project 元素是 Ant 构建文件的根元素，每个构建文件至少包含一个 project 元素。每个 project 元素可包含多个 target 元素。

译注3: target 元素为 Ant 的基本执行单元，它可以包含一个或多个具体的任务。多个 target 可以存在相互依赖关系。

译注4: Ant 支持许多可选任务，可选任务即需要加载额外的 java 类库方能执行的 Ant 任务。它们与 Ant 核心任务打包在一起。详情请参见：*http://ant.apache.org/manual/install.html#optionalTasks*。

虽然这里用 Apache Ant 来说明本章的核心概念，但还有许多其他工具也可用来构建 Web 应用。其中，值得一提的是 Rake（*http://rake.rubyforge.org/*），它已在近几年日趋流行。Rake 是一个基于 Ruby 的类 `make` 的构建程序。特别值得注意的是，Rakefiles（Makefiles 的 Rake 版）是用标准的 Ruby 语法编写的，因此具有平台无关性。

合并多个 JavaScript 文件

Combining JavaScript Files

根据 Yahoo!特别性能小组的研究，网站提速指南中第一条也是最重要的一条规则，就是减少页面渲染所需的 HTTP 请求数（*http://yuiblog.com/blog/2006/11/28/performance-research-part-1/*），特别是针对那些首次访问网站的用户。这是你优化工作的切入点，因为合并资源通常能以较少的工作为用户赢取最大的潜在利益。

大多数现代网站使用多个 JavaScript 文件：通常包括一个小型类库，它包含一系列工具和控件以简化跨浏览器的富交互 Web 应用开发；另外还有一些站点相关的代码，被分割成多个逻辑单元以便于开发者进行管理。例如 CNN（*http://www.cnn.com*），使用了 Prototype 和 Script.aculo.us 类库。它的首页有着 12 个外链脚本和 20 多段内嵌脚本。一个简单的优化方式是把部分文件和代码合并成一个外链文件，从而大大降低页面渲染所需的 HTTP 请求数。

Apache Ant 提供了合并多个文件的 `concat` 任务。但重要的是请注意，JavaScript 文件通常需要按照特定的依赖顺序连接在一起。依赖关系一旦确定，使用 `filelist` 或 `fileset` 元素集合能保存文件顺序。该 Ant target 示例如下：

```
<target name="js.concatenate">
    <concat destfile="${build.dir}/concatenated.js">
        <filelist dir="${src.dir}"
            files="a.js, b.js"/>
        <fileset dir="${src.dir}"
            includes="*.js"
            excludes="a.js, b.js"/>
    </concat>
</target>
```

该 target 在构建目录下创建一个名为 concatenated.js 的文件，它首先连接 a.js，紧接着是 b.js，然后是源目录下以字母顺序排列的其他文件。

请注意如果源文件中有任何一个文件（最后一个除外）不是以分号或行终止符结束的，那么

合并结果可能不是有效的 JavaScript 代码。这个问题可以通过给 concat 任务添加 `fixlastline` 属性来修复，这个属性指示 Ant 检查每个被合并的文件是否以换行符结尾：

```
<concat destfile="${build.dir}/concatenated.js" fixlastline="yes">
    ...
</concat>
```

预处理 JavaScript 文件

Preprocessing JavaScript Files

> 在计算机科学中，预处理器是在程序中处理输入数据，产生能用来输入到其他程序的数据的程序。输出被称为输入数据预处理过的形式，常用在后来的程序中，比如编译器。所处理的数量和种类依赖于预处理器的类型，一些预处理器只能执行相对简单的文本替换和宏扩展，而另一些则有着完全成熟的编程语言的能力。
>
> —*http://zh.wikipedia.org/zh/*预处理器

预处理你的 JavaScript 源文件并不会让应用变得更快，但它允许你做些其他的事情，例如有条件地插入测试代码，来衡量你的应用程序的性能。

由于没有专门为 JavaScript 设计的预处理器，所以必须使用一个词法预处理器，它要足够灵活，可定制其词法分析规则。或者使用一个为某种语言设计的工具，其词法语法与 JavaScript 的词法语法要足够接近。由于 C 语言的语法接近 JavaScript，因此 C 预处理器（`cpp`）是个不错的选择。应用 cpp 后的 Ant target 看起来如下：

```
<target name="js.preprocess" depends="js.concatenate">
    <apply executable="cpp" dest="${build.dir}">
        <fileset dir="${build.dir}"
            includes="concatenated.js"/>
        <arg line="-P -C -DDEBUG"/>
        <srcfile/>
        <targetfile/>
        <mapper type="glob"
            from="concatenated.js"
            to="preprocessed.js"/>
    </apply>
</target>
```

这个 target 依赖 `js.concatenate` target，它对前面的合并后文件运行 `cpp`，并在构建目录中创建 preprocessed.js 文件保存运行结果。注意 `cpp` 使用了标准的 `-P`（忽略供 C 编译器使用的行编号信息[译注5]）和 `-C`（包含程序注释）选项。在本例中，还定义了 `DEBUG` 宏。

有了这个 target，你现在可以直接在 JavaScript 文件中使用宏定义（**`#define, #undef`**）和条件编译（**`#if, #ifdef, #ifndef, #else, #elif, #endif`**）指令，比如，你可以使用条件语句启用（或删除）性能检测代码：

[译注5] 参见 *http://study.chyangwa.com/IT/AIX/aixcmds1/cpp.htm#a3z7zf35aclif*。

```
#ifdef DEBUG

(new YAHOO.util.YUILoader({
    require: ['profiler'],
    onSuccess: function(o) {
        YAHOO.tool.Profiler.registerFunction('foo', window);
    }
})).insert();

#endif
```

如果你打算使用多行宏，确保是使用 Unix 行结束符（LF）。你可以使用 `fixcrlf` Ant 任务自动帮你修复。

另一个例子，严格意义上与性能无关，但说明了 JavaScript 预处理如何强大，它使用了“variadic macros”（可接收多个参数的宏）和文件包含（file inclusion），以实现 JavaScript 断言。考虑下面的 include.js：

```
#ifndef _INCLUDE_JS_
#define _INCLUDE_JS_

#ifdef DEBUG
function assert(condition, message) {
    // 你可以在处理断言时弹出一个提示框，显示类似堆栈跟踪等信息。
}
#define ASSERT(x, ...) assert(x, ## __VA_ARGS__)
#else
#define ASSERT(x, ...)
#endif /* DEBUG */

#endif /* _INCLUDE_JS_ */
```

现在你可以像下面这样编写 JavaScript 代码：

```
#include "include.js"

function myFunction(arg) {
    ASSERT(YAHOO.lang.isString(argvar), "arg should be a string");
    ...
#ifdef DEBUG
    YAHOO.log("Log this in debug mode only");
#endif
    ...
}
```

断言和额外的日志代码只出现在开发过程的 `DEBUG` 宏块中。这些语句不会出现在最终产品中。

JavaScript 压缩

JavaScript Minification

JavaScript 压缩指的是把 JavaScript 文件中所有与运行无关的部分进行剥离的过程。剥离的内容包括注释和不必要的空白字符。该过程通常可以将文件大小减半，促使文件更快被下载，并鼓励程序员编写更好更详细的行内文档。

JSMin（*http://www.crockford.com/javascript/jsmin.html*），由 Douglas Crockford 开发，它在很长一段时间里被当作是 JavaScript 压缩的标准。然而，随着 Web 应用在规模和复杂性上不断增长，许多人认为应该把 JavaScript 压缩再向前推进一步。这也是开发 YUI Compressor（*http://developer.yahoo.com/yui/compressor/*）的主要原因，它提供了所有类型的智能操作，以一种绝对安全的方式提供了比其他工具更高级别的压缩。除了移除注释和不必要的空格，YUI Compressor 还提供如下功能：

- 将局部变量替换成更短的形式（1 个、2 个或 3 个字符），以优化后续的 Gzip 压缩工作。
- 尽可能将方括号表示法替换为点表示法（例如：`foo["bar"]` 被替换成 `foo.bar`）。
- 尽可能去掉直接量属性名的引号（例如：`{"foo" : "bar"}` 被替换成 `{foo : "bar"}`）。
- 替换字符串中的转义符号（例如：`'aaa\'bbb'`被替换成`"aaa'bbb"`）。
- 合并常量（例如：`"foo" + "bar"`被替换成`"foobar"`）。

相比 JSMin，你的代码经过 YUI Compressor 处理后会大幅缩减而无须更多操作。下面是 YUI 类库核心代码压缩前后的数据（版本 2.7.0，下载地址 *http://developer.yahoo.com/yui/*）：

```
原始版本 yahoo.js、dom.js 和 event.js           192 164 字节
yahoo.js, dom.js 和 event.js + JSMin           47 316 字节
yahoo.js, dom.js 和 event.js + YUI Compressor  35 896 字节
```

在这个例子中，YUI Compressor 相比 JSMin 节省了 24% 的空间。不过，你还可以做这些事情来进一步节省字节数：将局部引用存储在对象/值中、用闭包封装代码、使用常量替代重复值、避免 `eval`（以及类似的 `Function` 构造函数，使用 `setTimeout` 和 `setInterval` 时传递字符串直接量作为第一个参数）、`with` 关键字、JScript 条件注释都有助于进一步精简文件。考虑接下来的函数，它用于开启/关闭指定 DOM 元素的“`selected`”class（220 字节）：

```
function toggle (element) {
    if (YAHOO.util.Dom.hasClass(element, "selected")){
        YAHOO.util.Dom.removeClass(element, "selected");
    } else {
        YAHOO.util.Dom.addClass(element, "selected");
    }
}
```

YUI Compressor 将此代码转换如下（147 字节）：

```
function toggle(a){if(YAHOO.util.Dom.hasClass(a,"selected")){
YAHOO.util.Dom.removeClass(a,"selected")}else{YAHOO.util.Dom.
addClass(a,"selected")}};
```

如果你重构原始代码，将 `YAHOO.util.Dom` 存入一个局部引用，并使用常量存放`"selected"`值，代码将变成如下的样子（232 字节）：

```
function toggle (element) {
    var YUD = YAHOO.util.Dom, className = "selected";
    if (YUD.hasClass(element, className)){
        YUD.removeClass(element, className);
    } else {
        YUD.addClass(element, className);
    }
}
```

此版本经过 YUI Compressor 压缩后变得更小了（115 字节）：

```
function toggle(a){var c=YAHOO.util.Dom,b="selected";if(c.hasClass(a,b)){
c.removeClass(a,b)}else{c.addClass(a,b)}};
```

压缩率从 33%变为 48%，只需少量工作就得到惊人结果。然而，要注意后续的 Gzip 压缩可能会导致相反的结果；换句话说，最小的压缩文件并不意味着最小的 Gzip 压缩文件。这种奇怪的结果是降低源文件的冗余量造成的。此外，这类微优化（microoptimization）还导致了一个很小的运行期消耗，因为变量代替了直接量，从而需要额外的查找。因此，通常建议不要滥用这些技术，虽然当用户代理不支持（或只是声称支持）Gzip 压缩时，它们还是值得考虑的。

2009 年 11 月，Google 发布了一款更先进的压缩工具，名为“Closure Compiler”（*http://code.google.com/closure/compiler/*）。当使用高级优化选项时，该工具比 YUI Compressor 更为强大。在该模式下，Closure Compiler 以极富侵略性的方式转换代码并重命名符号。尽管它产生了难以置信的压缩率，但这要求开发者必须非常小心，以确保输出代码与输入代码以相当方式工作。它还让调试变得更麻烦，因为所有符号都被重命名了。Closure 类库倒是带有一个 Firebug 扩展，名为 Closure Inspector（*http://code.google.com/closure/compiler/docs/inspector.html*），它提供了混淆后符号与原始符号的映射。不过，这个扩展不能用于

Firefox 之外的浏览器，所以在调试浏览器特定的代码时这是个问题，而且与其他那些侵略性较弱压缩工具相比，调试工作依然更困难。

构建时处理对比运行时处理

Buildtime Versus Runtime Build Processes

合并、预处理和压缩这些步骤既可以在构建时进行，也可以在运行时进行。在开发过程中，运行时处理会非常有帮助。但由于扩展性原因一般不建议在产品环境中使用。开发高性能应用的一个普遍规则是，只要是能在构建时完成的工作，就不要留到运行时去做。

Apache Ant 无疑是一个离线构建程序，而本章末出现的敏捷构建工具则代表一种中间路线，同样的工具既可用在开发期间创建最终断言，也可用于生产环境。

JavaScript 的 HTTP 压缩

JavaScript Compression

当 Web 浏览器请求一个资源时，它通常会发送一个 `Accept-Encoding` HTTP 头（始于 HTTP/1.1）来告诉 Web 服务器它支持哪种编码转换类型。这个信息主要用来压缩文档以获得更快的下载，从而改善用户体验。`Accept-Encoding` 可用的值包括：`gzip`、`compress`、`deflate` 和 `identity`（这些值已经在 Internet Assigned Numbers Authority（简称 IANAA）注册）。

如果 Web 服务器在请求中看到这些信息头，它会选择最适合的编码方法，并通过 `Content-Encoding` HTTP 头通知 Web 浏览器它的决定。

`gzip` 是目前最流行的编码方式。它通常能减少 70% 的下载量，成为提升 Web 应用性能的首选武器。记住 Gzip 压缩主要适用于文本，包括 JavaScript 文件。其他类型，诸如图片或 PDF 文件，不应该使用 Gzip 压缩，因为它们本身已经被压缩过，试图重复压缩只会浪费服务器资源。

如果你使用 Apache Web 服务器（目前最流行的 Web 服务器），启用 Gzip 压缩需要安装并配置 `mod_gzip` 模块（适用于 Apache 1.3，详见 *http://www.schroepl.net/projekte/ mod_gzip/*）或 `mod_deflate` 模块（适用于 Apache 2）。

Yahoo!搜索和 Google 近期的独立研究表明，美国大型网站提供的内容中有大约 15%未经压缩。大多数是因为请求中丢失 `Accept-Encoding` HTTP 头，它被一些公司代理服务器、防火

墙、甚至是 PC 安全软件过滤掉了。尽管 Gzip 压缩对开发者而言是个神奇的工具，但必须注意到这个事实，尽量书写简洁的代码。另一种技术是提供替代的 JavaScript 内容让那些不能受益于 Gzip 压缩的用户能获得较为轻量级的使用体验（然而用户可以选择切换回完整版本）。

为此，值得一提的是 Packer（*http://dean.edwards.name/packer/*），一个由 Dean Edwards 开发的 JavaScript 压缩工具。Packer 压缩 JavaScript 文件的能力超过了 YUI Compressor。考虑下面关于 jQuery 类库的压缩结果（版本 1.3.2，位于 *http://www.jquery.com*）：

```
jQuery                          120 180 字节
jQuery + YUI Compressor          56 814 字节
jQuery + Packer                  39 351 字节
Raw jQuery + gzip                34 987 字节
jQuery + YUI Compressor + gzip19 457 字节
jQuery + Packer + gzip           19 228 字节
```

经过 Gzip 压缩后，jQuery 库用 Packer 和 YUI Compressor 压缩后的结果十分相似。然而，使用 Packer 压缩文件会造成一个固定的运行期消耗（在我的笔记本上是 200 到 300 毫秒）。因此使用 YUI Compressor 合并再用 Gzip 压缩是最佳选择。然而，Packer 可以用在网速较慢或不支持 Gzip 的情况下，解压缩的代价与下载大量代码的代价相比微不足道。为不同用户提供不同的 JavaScript 脚本的唯一确定就是会增加 QA[译注6] 成本。

缓存 JavaScript 文件

Caching JavaScript Files

缓存 HTTP 组件能极大提高网站回访用户的体验。举个具体的例子，在有缓存的情况下加载 Yahoo! 首页（*http://www.yahoo.com*）比没有缓存时的 HTTP 请求数要少 90%，下载的字节少 83%。往返时间（从请求页面开始到 onload 事件触发之前的这段时间）从 2.4 秒减少到 0.9 秒（*http://yuiblog.com/blog/2007/01/04/performance-research-part-2/*）。尽管缓存大多数用于图片，但它适用于所有静态内容，包括 JavaScript 文件。

Web 服务器通过“`Expires` HTTP 响应头”来告诉客户端一个资源应当缓存多长时间。它的值是一个遵循 RFC 1123 标准的绝对时间戳。例如：`Expires: Thu, 01 Dec 1994 16:00:00 GMT`。

[译注6] QA 意为“质量保证”，参见 *http://en.wikipedia.org/wiki/Quality_assurance*。

要将响应标记为“永不过期”，服务器可以发送一个时间为请求时间之后一年的 `Expires` 日期。按照 HTTP 1.1 RFC（RFC 2616，14.21 节），Web 服务器发送的 `Expires` 日期不应超过 1 年。

如果你使用 Apache web server，`ExpiresDefault` 指令允许你根据当前时间设置到期日期。下面的例子就把这个指令应用于图片、JavaScript 文件和 CSS 样式表：

```
<FilesMatch "\.(jpg|jpeg|png|gif|js|css|htm|html)$">
    ExpiresActive on
    ExpiresDefault "access plus 1 year"
</FilesMatch>
```

某些 Web 浏览器，特别是移动设备里的浏览器，可能会有缓存限制。例如，iPhone 上的 Safari 浏览器不会缓存解压后超过 25KB 的文件（参见 *http://yuiblog.com/blog/2008/02/06/iphone-cacheability/*），而在 iPhone 3.0 系统中限制为 15KB。在这种情况下，应该权衡 HTTP 组件数量和它们的可缓存性，考虑将它们分解成更小的块。

你也可以考虑在条件允许的情况下采用客户端存储机制，此时 JavaScript 代码必须自行控制到期时间。

最后，还有一种技术是使用 HTML5 离线应用缓存，它已在 Firefox 3.5、Safari 4.0 及 iPhone OS 2.1 以上版本的 Mobile Safari 上实现。这项技术依赖一个清单文件（manifest file），来列出所有待缓存的资源。这个清单文件通过向`<html>` 标签（记住使用 HTML5 的 DOCTYPE）中添加一个 `manifest` 属性来声明：

```
<!DOCTYPE html>
<html manifest="demo.manifest">
```

此清单文件使用一个特殊的语法列出离线资源，并且必须设置 mime type 为 `text/cache-manifest`。更多关于离线 Web 应用缓存的信息可访问 W3C 网站：*http://www.w3.org/TR/html5/offline.html*。

处理缓存问题

Working Around Caching Issues

适当的缓存控制能切实提升用户体验，但它有一个缺点：当应用升级时，你需要确保用户下载到最新的静态内容。这个问题可以通过把改动过的静态资源重命名来解决。

多数情况下，开发者会给文件增加一个版本或开发编号。有些人喜欢附加一个校验和[译注7]（checksum）。个人而言，我更偏向使用时间戳。这个任务能用 Ant 自动完成。接下来的例子演示了给 JavaScript 文件名附加格式为 `yyyyMMddhhmm` 的时间戳：

[译注7] 校验和（checksum）是一种数学算法，用于检验信息是否跟原始版本保持一致。

```
<target name="js.copy">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Rename JavaScript files by appending a time stamp -->
    <copy todir="${build.dir}">
        <fileset dir="${src.dir}" includes="*.js"/>
        <globmapper from="*.js" to="*-${DSTAMP}${TSTAMP}.js"/>
    </copy>
</target>
```

使用内容分发网络（CDN）

Using a Content Delivery Network

内容分发网络（CDN）是在互联网上按地理位置分布计算机网络，它负责传递内容给终端用户。使用 CDN 的主要原因是增强 Web 应用的可靠性、可扩展性，更重要的是提升性能。事实上，通过向地理位置最近的用户传输内容，CDN 能极大地减少网络延时。

一些大公司会维护自建的 CDN，但通常使用第三方 CDN 服务更经济一些，比如 Akamai Technologies（*http://www.akamai.com/*）或 Limelight Networks（*http://www.limelightnetworks.com/*）。

切换到 CDN 只需改写少量的代码，却很有可能明显提升终端用户的响应速度。

值得一提的是，绝大多数主流 JavaScript 类库都可通过 CDN 访问。例如，YUI 类库可直接从 Yahoo! 网络（服务器名为 *yui.yahooapis.com*，更多信息可访问 *http://developer.yahoo.com/yui/articles/hosting*）获得，jQuery、Dojo、Prototype、Script.aculo.us、MooTools、YUI 及其他类库都能直接通过 Google 的 CDN（服务器名为 *ajax.googleapis.com*，更多信息可访问 *http://code.google.com/apis/ajaxlibs/*）获得。

部署 JavaScript 资源

Deploying JavaScript Resources

部署 JavaScript 资源的过程通常需要复制大量文件到一台或多台远程主机，有时还需要在远程主机上执行一系列的 shell 命令，尤其是使用 CDN 时，需要通过传输网络分发新添加的文件。

Apache Ant 提供几个选项用于复制文件到远程主机。你可以使用 `copy` 任务复制文件到一个挂载到本地的文件系统，或者使用 `FTP` 或 `SCP` 任务。我个人偏好直接使用 `scp` 工具，因为所有主流平台都支持它。下面是一个非常简单的例子：

```
<apply executable="scp" failonerror="true" parallel="true">
    <fileset dir="${build.dir}" includes="*.js"/>
    <srcfile/>
    <arg line="${live.server}:/var/www/html/"/>
</apply>
```

最后，为了在运行 SSH 守护进程的远程主机上执行 Shell 命令，你可以使用可选的 `SSHEXEC` 任务或直接调用 ssh 工具，下面的例子演示了如何重启 Unix 主机上的 Apache web server：

```
<exec executable="ssh" failonerror="true">
    <arg line="${live.server}"/>
    <arg line="sudo service httpd restart"/>
</exec>
```

敏捷 JavaScript 构建过程

Agile JavaScript Build Process

传统的 build 工具很优秀，但大多数的 Web 开发人员都嫌它们太麻烦，因为每一次变更代码后都必须手动编译。Web 开发人员更中意的方案是跳过编译的步骤，只需刷新浏览器窗口即可。因此，很少有 Web 开发人员使用本章描述的技术，导致 Web 应用或网站的性能很糟糕。值得庆幸的是，编写这样一个结合这些先进技术的工具十分简单，它能帮助 Web 开发人员在高效工作的同时依然能获得他们的应用最佳性能。

`smasher` 是个 PHP5 编写的应用程序，基于 Yahoo!搜索使用的一个内部工具。它能合并多个 JavaScript 文件并预处理，根据选项对内容进行压缩。它能够以命令行的方式运行，或者在开发过程中处理 Web 请求并自动合并资源。源代码位于 *http://github.com/jlecomte/smasher*，包含以下文件：

`smasher.php`
核心文件

`smasher.xml`
配置文件

`smasher`
命令行封装

`smasher_web.php`
Web 服务访问入口

`smasher` 需要一个 XML 配置文件，它包含了待合并的文件组的定义，以及系统相关的各种信息。以下是这个文件的一个例子：

```
<?xml version="1.0" encoding="utf-8"?>
<smasher>
    <temp_dir>/tmp/</temp_dir>
    <root_dir>/home/jlecomte/smasher/files/</root_dir>
    <java_bin>/usr/bin/java</java_bin>
    <yuicompressor>/home/jlecomte/smasher/yuicompressor-2-4-2.jar</yuicompressor>

    <group id="yui-core">
        <file type="css" src="reset.css" />
        <file type="css" src="fonts.css" />
        <file type="js" src="yahoo.js" />
        <file type="js" src="dom.js" />
        <file type="js" src="event.js" />
    </group>

    <group id="another-group">
        <file type="js" src="foo.js" />
        <file type="js" src="bar.js" />
        <macro name="DEBUG" value="1" />
    </group>

    ...

</smasher>
```

每个 `group` 元素包含了一个 JavaScript 或 CSS 文件的集合。顶层元素 `root_dir` 包含查找这些文件的目录路径。`group` 元素还可以有选择地包含一系列预处理的宏定义。

配置文件保存后，你就可以从命令行运行 `smasher`。如果你不带任何参数运行它，它显示一些使用说明并退出。下面的例子演示了如何合并、预处理及压缩 YUI 类库的核心 JavaScript 文件：

```
$ ./smasher -c smasher.xml -g yui-core -t js
```

如果一切正常，可以在工作目录找到输出文件，它的名字以 group 的名字开头（本例为 `yui-core`），后面跟着一个时间戳和文件扩展名（例如，yui-core-200907191539.js）。

同样，你可以在开发过程中使用 `smasher` 处理 Web 请求，把 smasher_web.php 放在 Web 服务器根目录下，使用类似这样的 URL：

```
http://<host>/smasher_web.php?conf=smasher.xml&group=yui-core&type=css&nominify
```

通过在开发环境和生产环境为 JavaScript 和 CSS 资源使用不同的 URL，你现在可以无须理会构建过程，在高效工作的同时也获得最佳性能。

小结

Summary

构建与部署的过程对基于 JavaScript 的 Web 应用的性能有着巨大影响。这个过程中最重要的步骤有：

- 合并 JavaScript 文件以减少 HTTP 请求数。
- 使用 YUI Compressor 压缩 JavaScript 文件。
- 在服务器端压缩 JavaScript 文件（Gzip 编码）。

- 通过正确设置 HTTP 响应头来缓存 JavaScript 文件，通过向文件名增加时间戳来避免缓存问题[译注8]。
- 使用 CDN（Content Delivery Network）提供 JavaScript 文件；CDN 不仅可以提升性能，它也为你管理文件的压缩与缓存。

所有这些步骤都应该自动化处理，可以使用公开的工具，比如 Apache Ant，也可以使用定制的工具来满足你的特定需求。如果你使得构建过程工作起来，你将会显著提高那些依赖大量 JavaScript 的 Web 应用或网站的性能。

[译注8] 经过前面步骤的设置和浏览器本身的缓存机制，使得客户端存在静态文件缓存，于是服务端需要更新指定文件就变得麻烦。因此需通过改变文件名（实际上是为了改变该 HTTP 请求的 URI）的方式强制浏览器重新加载指定文件。

第 10 章

工具

Tools

Matt Sweeney

要找出脚本加载和运行时的性能瓶颈，适合的软件是必不可少的。许多浏览器厂商和大型网站分享了各种技术和工具，以推动 Web 变得更快、更高效。本章重点介绍其中的一些免费工具：

性能分析

在脚本运行期间定时执行各种函数和操作，找出需要优化的部分。

网络分析

检查图片、样式表和脚本的加载过程，以及它们对页面整体加载和渲染的影响。

当一个特定的脚本或应用程序没有达到最佳状态时，一个性能分析工具有助于安排优化工作的优先级。由于各种浏览器支持的程度不一样，这可能没那么简单，不过如今许多厂商已经在它们的调试工具中提供了性能分析工具。有些时候，性能问题可能与特定的浏览器相关，其他情况下，这类症状可能出现在多个浏览器中。请记住，在一个浏览器上进行的优化可能适用于其他浏览器，也可能产生相反的效果。性能分析工具确保优化的时间花费在系统最慢且影响大多数浏览器的地方，而不是去判定哪些函数或操作缓慢。

本章大多数内容专注于性能分析工具，但是网络分析工具也可以极大提高分析效率，以确保脚本和页面尽可能快地加载和运行。在埋头调整代码之前，你应该先确认所有的脚本和其他资源文件的加载过程已经被优化。图片和样式表的加载会影响脚本加载，这取决于浏览器允许的并发请求数和要加载的资源的数量。

其中的一些工具提供了如何提升网页性能的技巧。请记住，要想解释这些工具所提供的信

息最好先深入了解这些性能规则背后的基本原理。正如大多数规则那样，总会有例外发生，深入理解这些规则会让你知道什么时候应该打破它们。

JavaScript 性能分析

JavaScript Profiling

JavaScript 与生俱来的一个工具就是语言本身。使用 `Date` 对象可以测量脚本的任何部分。在其他工具出现前，测试脚本运行时间是一种常用手段，至今它依然偶尔会用到。`Date` 对象默认会返回当前时间，用一个 `Date` 实例减去另一个 `Date` 实例会得到以毫秒为单位的时间差。考虑下面的例子，它对比了通过模板创建元素与克隆已有元素所用的时间（参见第 3 章，DOM 编程）：

```
var start = new Date(),
    count = 10000,
    i, element, time;

for (i = 0; i < count; i++) {
    element = document.createElement('div');
}

time = new Date() - start;
alert('created ' + count + ' in ' + time + 'ms');

start = new Date();
for (i = 0, i < count; i++) {
    element = element.cloneNode(false);
}

time = new Date() - start;
alert('created ' + count + ' in ' + time + 'ms');
```

这种分析方法实现起来很不方便，它需要你手动插入计时代码。一个能处理时间计算并存储数据的 `Timer` 对象将会是一个更好的进阶方案。

```
Var Timer = {
    _data: {},

    start: function(key) {
        Timer._data[key] = new Date();
    },

    stop: function(key) {
        var time = Timer._data[key];
        if (time) {
            Timer._data[key] = new Date() - time;
        }
    },

    getTime: function(key) {
        return Timer._data[key];
        }
    };

    Timer.start('createElement');
    for (i = 0; i < count; i++) {
        element = document.createElement('div');
    }

    Timer.stop('createElement');
    alert('created ' + count + ' in ' + Timer.getTime('createElement');
```

正如你所看到的，这种方法仍需手动插入代码，但它提供了一个建立纯 JavaScript 性能分析的模式。通过扩展 `Timer` 对象的概念，一个性能分析工具可以在构造时注册函数并在计时代码中调用。

YUI Profiler

YUI Profiler（*http://developer.yahoo.com/yui/profiler/*）是一个用 JavaScript 编写的 JavaScript 性能分析工具，由 Nicholas Zakas 开发。除了计时功能，它还提供了针对函数、对象和构造器的性能分析接口，并提供性能分析数据的详细报告。它可以跨浏览器工作，输出的数据能提供更加强大的报告和分析。

YUI Profiler 提供了一个通用的定时器用于收集性能数据。它提供了一些静态方法，用于启动和停止已命名的定时器并获取性能数据。

```
var count = 10000, i, element;
    Y.Profiler.start('createElement');

    for (i = 0; i < count; i++) {
        element = document.createElement('div');
    }

    Y.Profiler.stop('createElement');

    alert('created ' + count + ' in  ' +
            Y.Profiler.getAverage('createElement') + 'ms');
```

相比之前提到的内置 `Date` 对象和 `Timer` 对象的方法，YUI Profiler 明显的改进在于提供了更多的性能数据，包括调用次数、平均时间、最小时间、最大时间等。这些数据可以收集起来与其他测试结果进行综合分析。

函数同样可以注册性能分析。注册过的函数会被植入收集性能数据的代码。例如，想要分析第 2 章提到的全局 `initUI` 方法，只需要传递它的函数名作为参数：

```
Y.Profiler.registerFunction("initUI");
```

许多函数被绑定在对象上以防止污染全局命名空间。对象的方法同样可以注册，只需把宿

主对象作为第二个参数传入 `registerFunction`。例如，假设一个 uiTest 对象实现了两个 `initUI` 方法，分别为 `uiTest.test1` 和 `uiTest.test2`，每个方法都可以独立注册：

```
Y.Profiler.registerFunction("test1", uiTest);
Y.Profiler.registerFunction("test2", uiTest);
```

这样工作得很好，但是靠它很难扩大范围去分析多个函数或整个应用程序。而 `registerObject` 方法会自动注册绑定在对象的每一个方法上：

```
Y.Profiler.registerObject("uiTest", uiTest);
```

第一个参数是对象的名字（用来生成报告），第二个参数是对象本身。它会把性能分析的代码植入 `uiTest` 对象的所有方法中。

那些依赖原型继承的对象需要特殊处理。YUI Profiler 允许注册构造函数，它会在对象的原型对象的所有方法中植入性能分析代码，所有对象实例都会生效：

```
Y.Profiler.registerConstructor("MyWidget", myNameSpace);
```

现在，`myNameSpace.MyWidget` 的每一个实例的每一个函数都会被分析并记入报告。你可以为每个注册的函数获取一个单独的报告对象：

```
var initUIReport = Y.Profiler.getReport("initUI");
```

它提供一个包含分析数据的对象，其中包括由每次调用消耗的时间值构成的数组，它们按照调用顺序排列。这些值可标绘出来用其他有趣的方法进行分析，检查时间上的变化。该对象有如下字段：

```
{
    min: 100,
    max: 250,
    calls: 5,
    avg: 120,
    points: [100, 200, 250, 110, 100]
};
```

有时候你可能只关心特定字段的值，YUI `Profiler` 有几个静态方法提供了每个函数或方法的单独数据：

```
var uiTest1Report = {
    calls: Y.Profiler.getCalls("uiTest.test1"),
    avg: Y.Profiler.getAvg("uiTest.test1")
};
```

要想正确地分析脚本的性能，必须要有一个能快速定位出代码中最慢部分的视图，有一个方法可以产生调用在某个对象或构造函数上的所有已注册函数的数据报告：

```
var uiTestReport = Y.Profiler.getReport("uiTest");
```

这会返回一个包含以下数据的对象：

```
{
    test1: {
        min: 100,
        max: 250,
        calls: 10,
        avg: 120
    },
    test2:
        min: 80,
        max: 210,
        calls: 10,
        avg: 90
    }
};
```

这提供了一个机会让你可以对数据进行排序并采用更有意义的方法查看数据，使得代码中速度较慢的部分受到更密切的关注。还有一个方法可以生成包含当前所有分析数据的完整报告。然而，它可能包含了许多无用信息，比如从未调用的函数，或那些性能已经达到预期指标的函数。为了尽量减少这些干扰，可传入一个函数来过滤这些数据：

```
var fullReport = Y.Profiler.getFullReport(function(data) {
    return (data.calls > 0 && data.avg > 5);
};
```

该函数返回的布尔值决定函数是否应该加入报告中，让不感兴趣的数据被过滤掉。

当分析完毕后，函数、对象和构造函数可以分别注销，清空分析数据：

```
Y.Profiler.unregisterFunction("initUI");
Y.Profiler.unregisterObject("uiTests");
Y.Profiler.unregisterConstructor("MyWidget");
```

`clear()`方法保留当前分析目标的注册状态，但会清空相关数据。函数可在每个函数或每次计时中单独调用：

```
Y.Profiler.clear("initUI");
```

如果不传入参数，那么所有数据都会被一次性清理：

```
Y.Profiler.clear();
```

由于分析报告是 JSON 格式，因此它能以多种方式查阅。最简单的查阅方法是输出 HTML 页面。还可以把它发送到服务器，存入数据库，实现更强大的报表功能。这在对比各种跨浏览器的优化技术时特别有用。

值得注意的是，匿名函数特别难以分析，因为它们没有名字。YUI Profiler 提供了一种机制对匿名函数进行植入，使得它可以被分析。注册一个匿名函数会返回一个封装函数，可以调用它来代替原来的匿名函数：

```
var instrumentedFunction =
    Y.Profiler.instrument("anonymous1", function(num1, num2){
        return num1 + num2;
    });
instrumentedFunction(3, 5);
```

这样就把匿名函数的数据添加到了 Profiler 的结果集中，而获取它的报告的方式与获取其他分析数据的方式相同：

```
var report = Y.Profiler.getReport("anonymous1");
```

匿名函数

Anonymous Functions

使用匿名函数或赋值函数（function assignments）可能会造成分析工具的数据变得混乱。许多被分析的函数都是匿名函数，因为在 JavaScript 里这是很常用的模式，但测量或分析它们很困难甚至无法进行。分析匿名函数的最佳办法是给它们取个名字。使用指针指向对象的方法而不是使用闭包，可以实现最充分的分析覆盖率。

比较两种方法，一个使用内联函数：

```
myNode.onclick = function() {
    myApp.loadData();
};
```

另一种使用方法调用：

```
myApp._onClick = function() {
    myApp.loadData();
};
myNode.onclick = myApp._onClick;
```

使用方法调用允许本章提到的任意一个分析工具自动追踪到 `onclick` 事件处理器。但这种方法不总是适合，因为它可能需要对代码进行大规模重构。

为了让分析工具自动追踪到匿名函数，增加一个内联名称会增加分析报告的可读性：

```
myNode.onclick = function myNodeClickHandler() {
myApp.loadData();
};
```

这种方法同样适用于声明为变量的函数（即函数表达式），有些分析工具在获取函数表达式的名称时会遇到麻烦：

```
var onClick = function myNodeClickHandler() {
    myApp.loadData();
};
```

现在匿名函数有了自己的名字，这使得大多数分析工具的分析结果能显示出有意义的结果。这些命名工作并不麻烦，甚至可以考虑在调试过程自动插入。

提示：尽量使用未压缩过的脚本来进行调试和性能分析。这样会确保你的函数易于识别。

Firebug

Firefox 是一款十分受开发者喜爱的浏览器，部分原因是 Firebug 插件（网址：*http:/www.getfirebug.com*），它最初由 Joe Hewitt 开发，如今由 Mozilla 基金会维护。Firebug 有着前所未有的代码洞察力，它提高了全世界 Web 开发人员的生产力。

Firebug 提供了一个控制台，它用来输出日志、当前页面的 DOM 树、显示样式信息、检查 DOM 和 JavaScript 对象的内部结构、以及更多功能。它还包含了一个性能与网络分析工具，这是本节讨论的重点。Firebug 还具有高度可扩展性，可以很容易地添加自定义面板。

控制台面板分析工具

Console Panel Profiler

Firebug 的分析工具是控制台面板的一部分（参见图 10-1）。它测量并报告页面中运行的 JavaScript。当分析工具运行时，它会报告每个被调用函数的细节，提供高度精确的性能数据和查看变量功能，有助于找出导致脚本变慢的潜在原因。

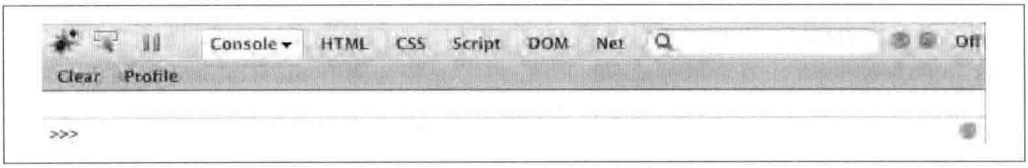
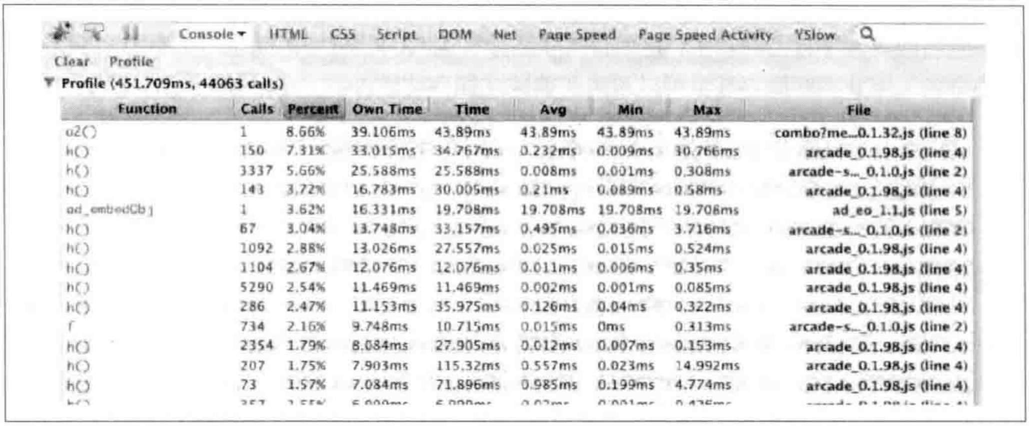


图 10-1　Firebug 控制台面板

点击“Profile”按钮即开始分析，触发脚本，再次点击“Profile”按钮可停止分析。图 10-2 显示了一个典型的数据分析报告。其中包括 Calls，表示函数被调用的次数；OwnTime，表示函数自身消耗的时间；Time，表示函数以及被它调用的函数所消耗的总时间。性能分析是浏览器的内核级处理过程，所以从控制台面板启动分析的性能开销非常小。



Profile (451.709ms, 44063 calls)

Function	Calls	Percent	Own Time	Time	Avg	Min	Max	File
o2()	1	8.66%	39.106ms	43.89ms	43.89ms	43.89ms	43.89ms	combo?me...0.1.32.js (line 8)
h()	150	7.31%	33.015ms	34.767ms	0.232ms	0.009ms	30.766ms	arcade_0.1.98.js (line 4)
h()	3337	5.66%	25.588ms	25.588ms	0.008ms	0.001ms	0.308ms	arcade-s..._0.1.0.js (line 2)
h()	143	3.72%	16.783ms	30.005ms	0.21ms	0.089ms	0.58ms	arcade_0.1.98.js (line 4)
ad_embedObj	1	3.62%	16.331ms	19.708ms	19.708ms	19.708ms	19.708ms	ad_eo_1.1.js (line 5)
h()	67	3.04%	13.748ms	33.157ms	0.495ms	0.036ms	3.716ms	arcade-s..._0.1.0.js (line 2)
h()	1092	2.88%	13.026ms	27.557ms	0.025ms	0.015ms	0.524ms	arcade_0.1.98.js (line 4)
h()	1104	2.67%	12.076ms	12.076ms	0.011ms	0.006ms	0.35ms	arcade_0.1.98.js (line 4)
h()	5290	2.54%	11.469ms	11.469ms	0.002ms	0.001ms	0.085ms	arcade_0.1.98.js (line 4)
h()	286	2.47%	11.153ms	35.975ms	0.126ms	0.04ms	0.322ms	arcade_0.1.98.js (line 4)
f	734	2.16%	9.748ms	10.715ms	0.015ms	0ms	0.313ms	arcade-s..._0.1.0.js (line 2)
h()	2354	1.79%	8.084ms	27.905ms	0.012ms	0.007ms	0.153ms	arcade_0.1.98.js (line 4)
h()	207	1.75%	7.903ms	115.32ms	0.557ms	0.023ms	14.992ms	arcade_0.1.98.js (line 4)
h()	73	1.57%	7.084ms	71.896ms	0.985ms	0.199ms	4.774ms	arcade_0.1.98.js (line 4)

图 10-2 Firebug 性能分析面板

Console API

Firebug 还提供了一个启动或停止性能分析的 JavaScript 接口。它让我们能更加精确地控制测量某部分代码。它还提供给报告命名的选项，这在比较多种优化技术时特别有价值。

```
console.profile("regexTest");
regexTest('foobar', 'foo');
console.profileEnd();
console.profile("indexOfTest");
indexOfTest('foobar', 'foo');
console.profileEnd();
```

在关注点上启动和停止分析工具可以减少副作用和其他脚本造成的干扰。有一点要记住，用这种方法调用分析工具会增加脚本的开销，主要是因为调用 `profileEnd()` 需要花费时间来生成报告，它会阻塞后续执行直到报告生成完毕。较大的报告需要更长的时间来生成，更好的做法是将 `profileEnd()` 调用封装在 `setTimeout` 中，使得报告的生成过程可以异步进行，从而不会阻塞脚本执行。

提示：此 JavaScript 接口同样可以在 Firebug 控制台的命令行窗口中调用。

分析完毕后，会生成一个新的报告，其中显示了每个函数所花的时间，调用的次数，占总开

销的百分比，以及其他有趣的数据。这些数据为判断接下来应该花功夫优化函数速度还是优化调用次数提供了依据。

类似 YUI Profiler，Firebug 的 `console.time()`函数有助于测量性能分析工具不会监测的循环和其他操作。例如，下面对一小段包含循环的代码进行计时：

```
console.time("cache node");
for (var box = document.getElementById("box"),
        i = 0;
    i < 100; i++) {
   value = parseFloat(box.style.left) + 10;
   box.style.left = value + "px";
}
console.timeEnd("cache node");
```

在定时器结束后，时间会被输出到控制台。这在对比多种优化方法时非常有意义。控制台还可以捕获并记录另外的计时结果，因此很容易并排显示分析结果。例如，想要比较缓存节点的引用和缓存节点样式对象的引用的性能区别，只需要把另一个版本的实现代码包在计时代码中即可：

```
console.time("cache style");
for (var style = document.getElementById("box").style,
        i = 0;
    i < 100; i++) {
   value = parseFloat(style.left) + 10;
   style.left = value + "px";
}
console.timeEnd("cache style");
```

控制台 API 使得程序员能够灵活地植入多层次的分析代码，并将结果汇总在报告中，然后使用各种有趣的方法进行分析。

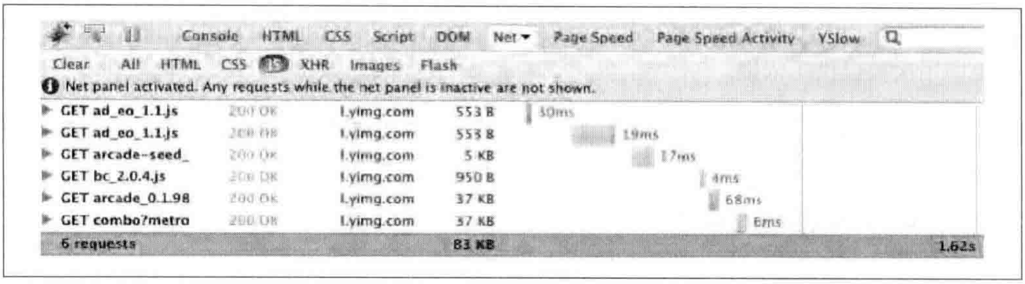
提示：点击一个函数会显示它所在的源文件的上下文，这在处理匿名或名字被混淆的函数时非常有用。

网络面板

Net Panel

往往在遇到性能问题时，最好从代码中抽身出来，站在更高的角度去观察。Firebug 在网络面板中提供了一个网络资源视图（参见图 10-3）。这个面板以可视化的方式展示了脚本对其他资源的阻塞作用，可以深入探查脚本对其他文件加载造成的影响，以及对页面的一般影响。

每个资源后面的色块条将加载过程分解为不同阶段（DNS 查找、等待响应等）。第一条垂直



```
Net panel activated. Any requests while the net panel is inactive are not shown.
GET ad_eo_1.1.js      200 OK   l.yimg.com   553 B   30ms
GET ad_eo_1.1.js      200 OK   l.yimg.com   553 B   19ms
GET arcade-seed_      200 OK   l.yimg.com    5 KB   17ms
GET bc_2.0.4.js       200 OK   l.yimg.com   950 B   4ms
GET arcade_0.1.98     200 OK   l.yimg.com   37 KB   68ms
GET combo?metro       200 OK   l.yimg.com   37 KB   6ms
6 requests                                  83 KB            1.62s
```

图 10-3 Firebug 网络面板

线（蓝色）表示页面 `DOMContentLoaded` 事件触发的时刻。这个事件标志着页面的 DOM 树已经解析完成并准备就绪。第二条垂直线（红色）表示 window 的 `load` 事件触发的时刻，它意味着 DOM 准备就绪后所有外部资源也已加载完成。这两条线展示了解析和执行所消耗的时间与页面渲染时间的对比。

正如你在图中看到的，有大量的脚本被下载。从时间线来看，每个脚本看上去都在等待前一个脚本下载完后才发起下一个请求。提高加载性能最简单的办法就是减少请求数，尤其是脚本和样式表文件的请求，因为它们可能会阻塞其他资源以及页面渲染。如果可能的话，把所有的脚本合并成一个文件，以最小化总请求数。这种方法同样适用于样式表和图片文件。

IE 开发人员工具[译注1]

Internet Explorer Developer Tools

自 IE 8 开始，IE 提供了一个开发工具包，其中包含了一个探查器[译注2]。该工具包内建于 IE 8 中，因此无须额外地下载或安装。类似 Firebug，IE 分析工具包括了函数分析，并提供一个包括调用次数、消耗时间、以及其他性能数据的详细报告。该报告能以调用树（call tree）的方式查看，还能分析原生函数，并导出分析数据。尽管它缺少网络分析工具，但是这个探查器可以配合其他通用工具，诸如 Fiddler 一起使用，它将在本章后面讨论。更多详细信息请参见 *http://msdn.microsoft.com/en-us/library/dd565628(VS.85).aspx*。

IE 8 的探查器可以从开发工具菜单项（工具→开发工具）启动。在按下“Start Profiling”[译注3]按钮后，所有随后的 JavaScript 活动都会被监视并分析。点击“Stop Profiling”（同一个的按钮，文字改变了）会停止探查器，并生成一个新的报告。默认快捷键是 F5 启动，Shift - F5 停止。

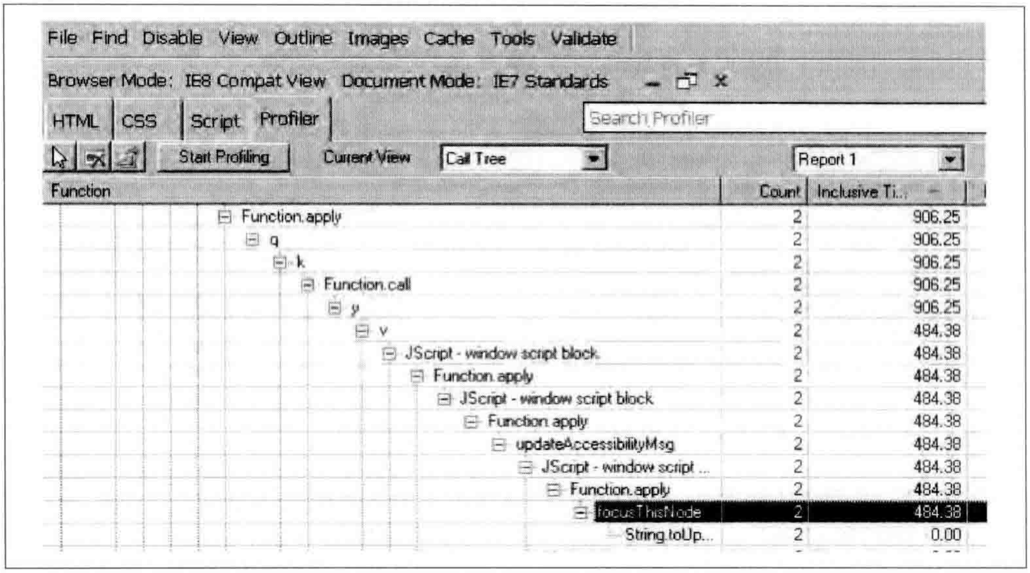
该报告提供了一个平面的函数视图，其中包括总时间和每次调用所耗的时间，另外还有一个树状图视图列出了函数调用栈。这个树状视图允许你逐级查看调用栈并定位运行慢的代码

[译注1] 以下名词参照 IE 8 中文版。

[译注2] 原文 Profiler，即分析工具，在 IE 8 中文版中名为“探查器”。

[译注3] IE 8 中文版中按钮文字汉化为“开始配置文件”，这应该是汉化失误。IE 9 中已将文字更正为“开始采样”。

的调用路径（参见图 10-4）。当函数的名字无法取得时，IE 探查器将以函数的变量名来显示。


File Find Disable View Outline Images Cache Tools Validate
Browser Mode: IE8 Compat View Document Mode: IE7 Standards
HTML CSS Script Profiler
Search Profiler
Start Profiling
Current View
Call Tree
Report 1

Function	Count	Inclusive Ti...
Function.apply	2	906.25
q	2	906.25
k	2	906.25
Function.call	2	906.25
y	2	906.25
v	2	484.38
JScript - window script block	2	484.38
Function.apply	2	484.38
JScript - window script block	2	484.38
Function.apply	2	484.38
updateAccessibilityMsg	2	484.38
JScript - window script ...	2	484.38
Function.apply	2	484.38
focusThisNode	2	484.38
String.toUp...	2	0.00

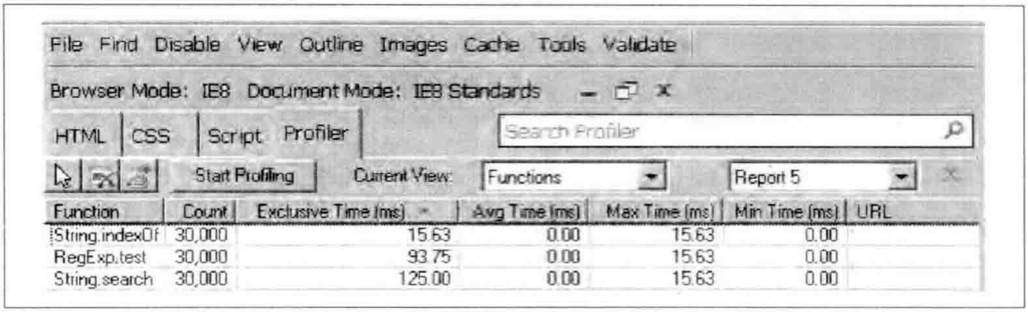
图 10-4　IE 8 探查器调用树

提示：右击分析结果表格可以添加或删除列。

IE 探查器还可以观察原生 JavaScript 对象的方法。除了分析实现代码外，你也能分析原生对象使得你能做一些这样的事情，比如比较 `String::indexOf` 和 `RegExp::test` 哪个更适合判断 HTML 元素的 `className` 是否以特定值开头：

```
var count = 10000,
    element = document.createElement('div'),
    result, i, time;

element.className  = 'foobar';

for (i = 0; i < count; i++) {
    result = /^foo/.test(element.className);
}

for (i = 0; i < count; i++) {
    result = element.className.search(/^foo/);
}

for (i = 0; i < count; i++) {
    result = (element.className.indexOf('foo') === 0);
}
```

如图 10-5 所示，不同的方法所用的时间差异很大。请记住，每次调用的平均时间为零。原生方法通常是最后需要优化的，但对比各种方法是个有趣的实验。同时请记住，由于这个数字很小，可能由于舍入误差和系统内存波动而无法得出确切结论。


File
Find
Disable
View
Outline
Images
Cache
Tools
Validate
HTML
CSS
Script
Profiler
Search Profiler
Start Profiling

Browser Mode: IE8 Document Mode: IE8 Standards

Current View: Functions

Report 5

Function	Count	Exclusive Time (ms)	Avg Time (ms)	Max Time (ms)	Min Time (ms)	URL
String.indexOf	30,000	15.63	0.00	15.63	0.00	
RegExp.test	30,000	93.75	0.00	15.63	0.00	
String.search	30,000	125.00	0.00	15.63	0.00	

图 10-5　原生方法的分析结果

尽管 IE 探查器目前不提供 JavaScript API，但它有一个能记录日志的控制台 API。可以将 `console.time()`和 `console.timeEnd()`函数从 Firebug 上移植过来，从而在 IE 上进行同样的测试。

```
if (console && !console.time) {
    console._timers = {};
    console.time = function(name) {
        console._timers[name] = new Date();
    };
    console.timeEnd = function(name) {
        var time = new Date() - console._timers[name];
        console.info(name + ': ' + time + 'ms');
    };
}
```

提示：IE 8 探查器的分析结果可以通过“导出数据”按钮导出为 .csv 格式。

Safari Web 检查器（Web Inspector）

Safari Web Inspector

Safari 从 4.0 版本开始，除了其他工具外，还提供了一个性能分析工具，包含在网络 analyzer 中，作为 Web Inspector 的一部分。类似 Internet Explorer Developer Tools，Web Inspector 能分析原生函数并提供一个可展开的调用树。它也包含了与 Firebug 类似的带分析功能的控制台 API，以及一个用于网络分析的资源面板。

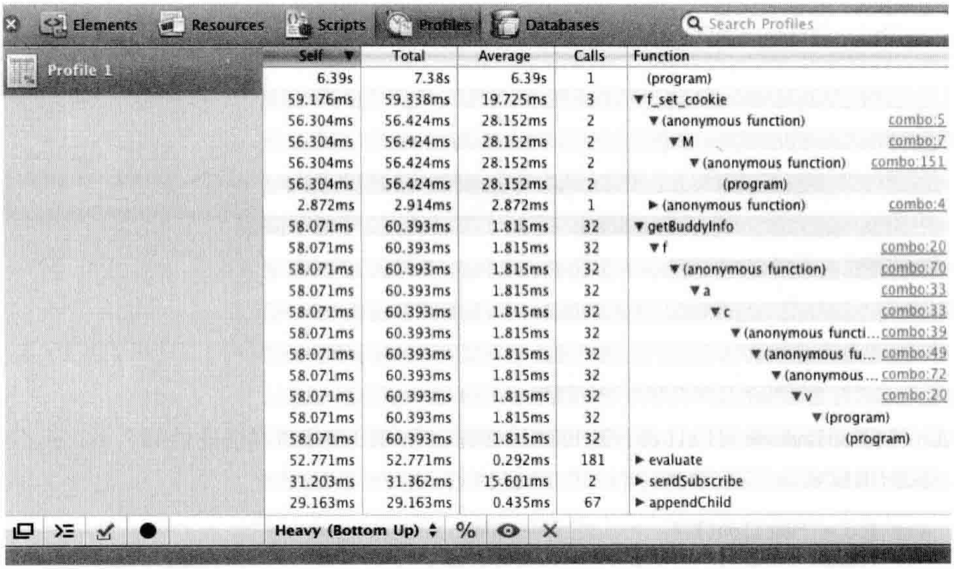
要访问 Web Inspector，首先确认“开发”菜单是否存在。“开发”菜单可用如下方法启用：打开 偏好设置→高级，勾选“在菜单栏中显示‘开发’菜单选项”。然后 Web 检查器就可以从开发→显示 Web 检查器（或键盘快捷键 Option-Command-I [译注4]）启动。

分析面板[译注5]

Profiles Panel

点击“Profile”按钮会出现 Profile 面板（如图 10-6）。点击“Enable Profiling”按钮即可启用。点击“Start Profiling”按钮（左下角的暗色圆形钮）即可开始分析。点击“Stop Profiling”（同一个按钮，此时是红色）可停止分析并显示报告。

提示：同样可以按下 Option-Shift-Command-P 来开始或停止分析。



Self ▼	Total	Average	Calls	Function	
6.39s	7.38s	6.39s	1	(program)	
59.176ms	59.338ms	19.725ms	3	▼ f_set_cookie	
56.304ms	56.424ms	28.152ms	2	▼ (anonymous function)	combo:5
56.304ms	56.424ms	28.152ms	2	▼ M	combo:7
56.304ms	56.424ms	28.152ms	2	▼ (anonymous function)	combo:151
56.304ms	56.424ms	28.152ms	2	(program)	
2.872ms	2.914ms	2.872ms	1	▶ (anonymous function)	combo:4
58.071ms	60.393ms	1.815ms	32	▼ getBuddyInfo	
58.071ms	60.393ms	1.815ms	32	▼ f	combo:20
58.071ms	60.393ms	1.815ms	32	▼ (anonymous function)	combo:70
58.071ms	60.393ms	1.815ms	32	▼ a	combo:33
58.071ms	60.393ms	1.815ms	32	▼ c	combo:33
58.071ms	60.393ms	1.815ms	32	▼ (anonymous functi...	combo:39
58.071ms	60.393ms	1.815ms	32	▼ (anonymous fu...	combo:49
58.071ms	60.393ms	1.815ms	32	▼ (anonymous...	combo:72
58.071ms	60.393ms	1.815ms	32	▼ v	combo:20
58.071ms	60.393ms	1.815ms	32	▼ (program)	
58.071ms	60.393ms	1.815ms	32	(program)	
52.771ms	52.771ms	0.292ms	181	▶ evaluate	
31.203ms	31.362ms	15.601ms	2	▶ sendSubscribe	
29.163ms	29.163ms	0.435ms	67	▶ appendChild	

图 10-6　Safari Web 检查器的分析面板

Safari 也模仿 Firebug 提供了 JavaScript API（`console.profile()`、`console.time()`等），以便可编程地启动和停止分析功能。此功能与 Firebug 相同，允许你对报告和计时进行命名以提供更好的分析管理。

[译注4] 此快捷键为 Mac 版，Windows 版对应的快捷键为：Ctrl+Alt+I。

[译注5] Safari 中文版中将 Profiles 翻译为“描述文件”，译者认为这是同 IE 开发工具条一样的汉化失误。所以后续的译文中按钮的文字直接保留英文。

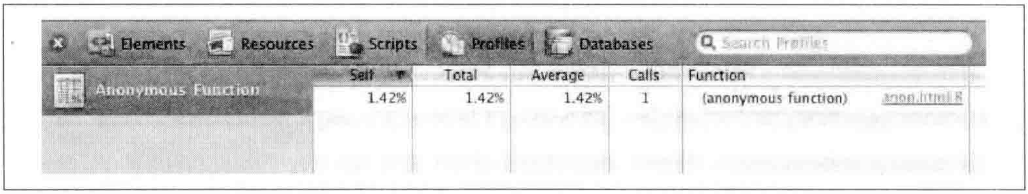
提示：`console.profileEnd()`还可以传入一个名字作为参数。它能停止指定的分析进程，如果有多个分析过程在同时进行的话。

Safari 同时提供一个 Heavy（bottom-up）视图用于分析函数，和一个 Tree（top-down）用于显示调用栈。默认的 Heavy 视图将最慢的函数排在前面，并允许遍历调用栈，而树视图至上而下地显示了从最外层调用开始的代码运行路径。分析调用树有助于揭示与函数调用方式相关的性能问题。

Safari 还增加了一个名为 `displayName` 的属性来帮助更好地进行分析。它提供了一种方法，可以为输出报告中的匿名函数命名。考虑下面这个赋值给变量 `foo` 的函数：

```
var foo = function() {
    return 'foo!';
};

console.profile('Anonymous Function');
foo();
console.profileEnd();
```

如图 10-7 所示，由于缺少函数名，最终的分析报告很难理解。点击函数右侧的 URL 可以显示该函数在源代码中的上下文。



Self	Total	Average	Calls	Function
1.42%	1.42%	1.42%	1	(anonymous function) anon.html:8

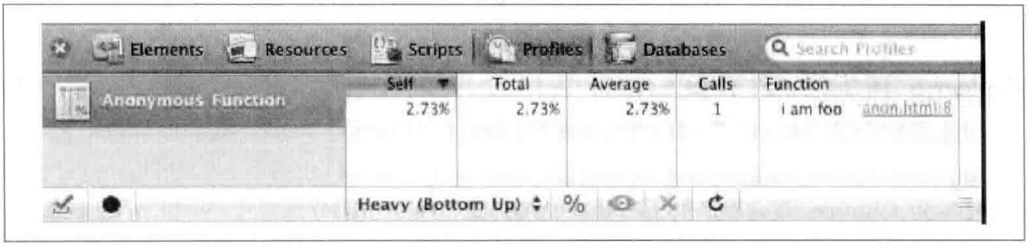
图 10-7 Web 检查器分析面板显示匿名函数

添加一个 `displayName` 可以让报告的可读性更好。你可以使用更具描述性的名字，而不仅限于合法的函数名。

```
var foo = function() {
    return 'foo!';
};
foo.displayName = 'I am foo';
```

如图 10-8 所示，`displayName` 的值现在取代了匿名函数字样。但是，这个属性仅适用于基于 Webkit 内核的浏览器。它还要求重构真正的匿名函数，所以不建议这样做。正如前文讨论过的，添加内联名称是命名匿名函数最简单的方法，而且可以兼容其他分析工具：

```
var foo = function foo() {
    return 'foo!';
};
```


Elements
Resources
Scripts
Profiles
Databases
Search Profiles
Anonymous Function
Heavy (Bottom Up)
%

Self	Total	Average	Calls	Function
2.73%	2.73%	2.73%	1	i am foo anon.html:8

图 10-8　Web 检查器分析面板显示了 displayName

资源面板

Resources Panel

资源面板能帮助你更好地理解 Safari 是如何加载和解析脚本以及其他外部资源的。类似 Firebug 的网络面板，它提供了一个资源视图，显示每一个发起的请求所消耗的时间。资源以不同颜色标记以方便查看。Web 检查器的资源面板将尺寸图表和时间图表分开展示，减少了视觉干扰（参见图 10-9）。

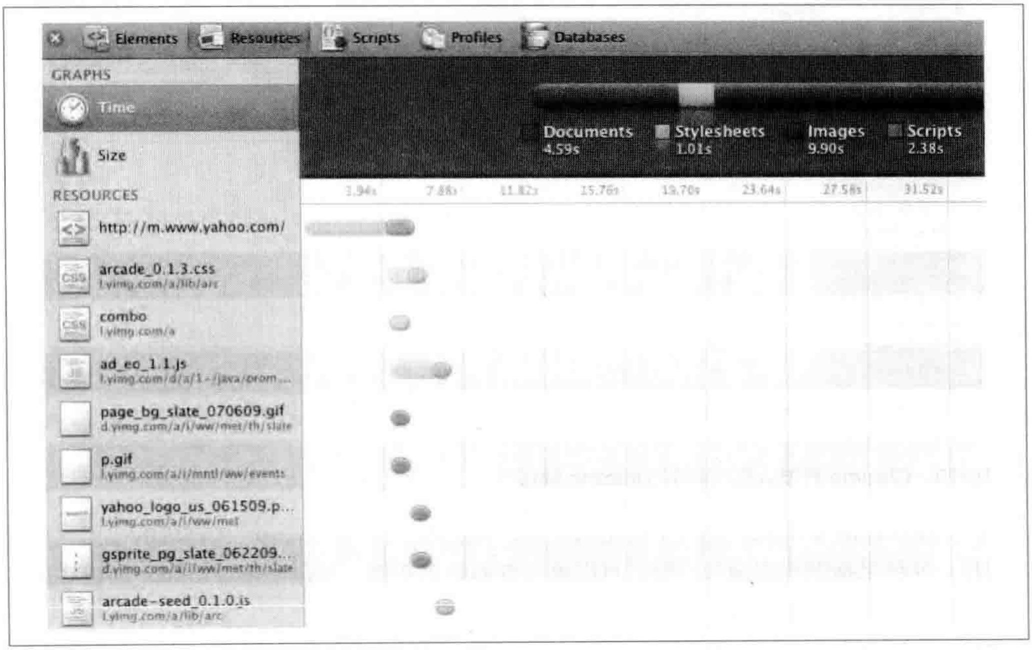

Elements
Resources
Scripts
Profiles
Databases
GRAPHS
Time
Size
RESOURCES
http://m.www.yahoo.com/
arcade_0.1.3.css
l.yimg.com/a/lib/arc
combo
l.yimg.com/a
ad_eo_1.1.js
l.yimg.com/d/a/1-/java/prom...
page_bg_slate_070609.gif
d.yimg.com/a/i/ww/met/th/slate
p.gif
l.yimg.com/a/i/mntl/ww/events
yahoo_logo_us_061509.p...
l.yimg.com/a/i/ww/met
gsprite_pg_slate_062209...
d.yimg.com/a/i/ww/met/th/slate
arcade-seed_0.1.0.js
l.yimg.com/a/lib/arc
Documents 4.59s
Stylesheets 1.01s
Images 9.90s
Scripts 2.38s
3.94s
7.88s
11.82s
15.76s
19.70s
23.64s
27.58s
31.52s

图 10-9　Safari 的资源面板

请注意，与其他浏览器不同的是，Safari 4 能够并行加载脚本而不会造成阻塞。Safari 通过确保脚本按照正确的顺序执行来绕开脚本阻塞的要求。请记住，这只适用于加载那些最初就嵌入 HTML 里的脚本，动态添加的脚本不会阻塞加载和执行（参见第 1 章）。

Chrome 开发人员工具

Chrome Developer Tools

Google 为 Chrome 浏览器提供了一套开发工具集，其中一部分是基于 WebKit/Safari Web Inspector。除监视网络流量的资源面板外，Chrome 还为所有页面和网络事件添加了一个时间线(timeline)视图[译注6]。Chrome 包含了 Web Inspector 分析面板以外，还添加了当前内存的“堆（heap）”快照功能。同 Safari 一样，Chrome 可以分析原生函数并实现了 Firebug 的控制台 API，其中包括 `console.profile` 和 `console.time`。

如图 10-10 所示，Timeline 面板提供了所有活动的概况，按分类可分为：“Loading”、“Scripting”或“Rendering”[译注7]。这使得开发人员可以快速定位系统中速度最慢的部分。某些事件包含其他事件行的子树（subtree），可以在报告视图中展开或隐藏以显示更多或更少的信息。

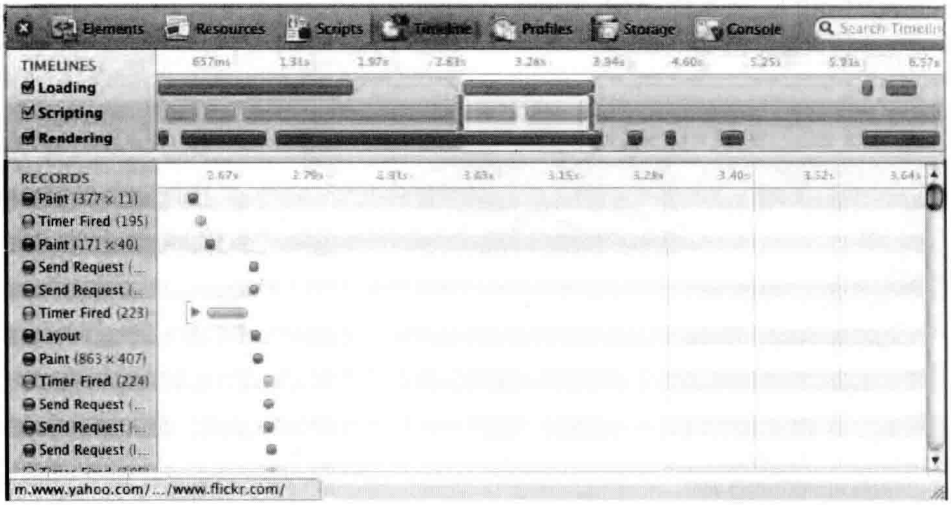

Elements
Resources
Scripts
Timeline
Profiles
Storage
Console
Search Timeline
TIMELINES
☑ Loading
☑ Scripting
☑ Rendering
657ms
1.31s
1.97s
2.63s
3.28s
3.94s
4.60s
5.25s
5.91s
6.57s
RECORDS
2.67s
2.79s
2.91s
3.03s
3.15s
3.28s
3.40s
3.52s
3.64s
Paint (377 × 11)
Timer Fired (195)
Paint (171 × 40)
Send Request (...
Send Request (...
Timer Fired (223)
Layout
Paint (863 × 407)
Timer Fired (224)
Send Request (...
Send Request (...
Send Request (l...
m.www.yahoo.com/.../www.flickr.com/

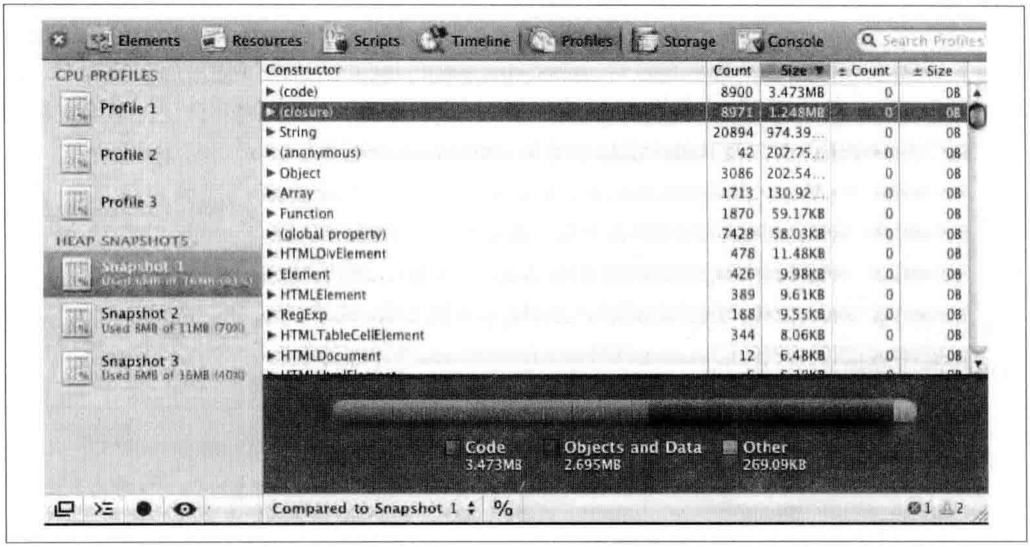
图 10-10　Chrome 开发人员工具的 Timeline 面板

点击 Chrome 的 Profiles 面板上的眼睛图标，可获得当前 JavaScript 内存堆的快照（见图 10-11）。其结果按照构造函数分组，可以展开查看每个实例。可以使用 Profiles 面板底部的

[译注6] Safari 5.0 版本的 Web Inspector 工具中也增加了 Timeline 面板，提供的功能大致相同。

[译注7] 以谷歌浏览器（中文版）为准，因此保留原文。

“Compared to Snapshot”选项来对比多个快照。带 +/－号的 Count 列和 Size 列显示出快照之间的差异。


Elements
Resources
Scripts
Timeline
Profiles
Storage
Console
Search Profiles
CPU PROFILES
Profile 1
Profile 2
Profile 3
HEAP SNAPSHOTS
Snapshot 1
Snapshot 2
Used 8MB of 11MB (70%)
Snapshot 3
Used 6MB of 16MB (40%)
Code
3.473MB
Objects and Data
2.695MB
Other
269.09KB
Compared to Snapshot 1
%
1
2

Constructor	Count	Size ▼	± Count	± Size
▶ (code)	8900	3.473MB	0	0B
▶ (closure)	8971	1.248MB	0	0B
▶ String	20894	974.39...	0	0B
▶ (anonymous)	42	207.75...	0	0B
▶ Object	3086	202.54...	0	0B
▶ Array	1713	130.92...	0	0B
▶ Function	1870	59.17KB	0	0B
▶ (global property)	7428	58.03KB	0	0B
▶ HTMLDivElement	478	11.48KB	0	0B
▶ Element	426	9.98KB	0	0B
▶ HTMLElement	389	9.61KB	0	0B
▶ RegExp	188	9.55KB	0	0B
▶ HTMLTableCellElement	344	8.06KB	0	0B
▶ HTMLDocument	12	6.48KB	0	0B

图 10-11　Chrome 开发人员工具的 JavaScript 堆快照

脚本阻塞
Script Blocking

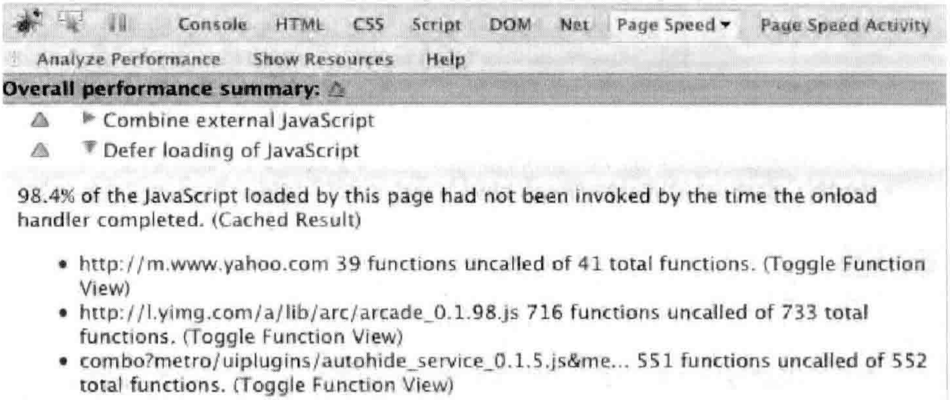
传统上，浏览器限制每次只能发出一个脚本请求。这样做是为了管理文件之间的依赖关系。只要一个文件依赖于在源码中靠后的另一个文件，那么它所依赖的那个文件必须保证在它运行之前准备就绪。（在分析工具的时间线视图中，）脚本之间存在间隙就说明脚本被阻塞了。新版浏览器诸如 Safari 4、IE 8、Firefox 3.5 以及 Chrome 解决这个问题的办法是允许并行下载，但阻塞运行，以保证依赖关系已经准备好。虽然这样做能使文件下载得更快，但页面渲染仍然会被阻塞，直到所有脚本都被执行。

脚本阻塞会因一个或多个文件初始化缓慢而变得更加严重，有必要对它做一些分析，并有可能需要优化或重构。脚本的加载会减慢甚至停止页面渲染，造成用户等待。网络分析工具有助于找出并优化加载资源之间的间隙。用可视化的图形描述这些脚本传输过程中的间隙，可以找出那些运行较慢的脚本。这些脚本或许应该推迟到页面渲染完成之后再加载，或者尽可能优化或重构以减少执行时间。

Page Speed

Page Speed 最初是 Google 为内部用途所开发的一个工具，后来作为 Firebug 的插件发布，它类似 Firebug 的网络面板，提供了 Web 页面加载的资源的信息。然而，除了加载时间和 HTTP 状态，它还能显示解析和运行 JavaScript 消耗的时间，指明可延迟加载的脚本，并报告那些没有被使用的函数。这些有价值的信息有助于为进一步的研究、优化、甚至重构指明方向。关于安装说明以及其他产品信息参见 *http://code.google.com/speed/page-speed/*。

Page Speed 面板上的“Profile Deferrable JavaScript”选项，指出哪些文件可以被延迟加载或拆分为一个较小片段以减少初始负载。通常来说，渲染初始视图需要运行的脚本很少。从图 10-12 中你可以看到，大部分已加载的代码在 window 的 load 事件触发前都不会用到。延迟加载这些无须立刻使用的代码可以让初始页面加载更快。脚本和其他资源可以根据需要有选择地延迟加载。


Console
HTML
CSS
Script
DOM
Net
Page Speed ▾
Page Speed Activity
Analyze Performance
Show Resources
Help
Overall performance summary: △
△ ▸ Combine external JavaScript
△ ▾ Defer loading of JavaScript

98.4% of the JavaScript loaded by this page had not been invoked by the time the onload handler completed. (Cached Result)

- http://m.www.yahoo.com 39 functions uncalled of 41 total functions. (Toggle Function View)
- http://l.yimg.com/a/lib/arc/arcade_0.1.98.js 716 functions uncalled of 733 total functions. (Toggle Function View)
- combo?metro/uiplugins/autohide_service_0.1.5.js&me... 551 functions uncalled of 552 total functions. (Toggle Function View)

图 10-12　Page Speed 对“可延迟加载的 JavaScript”的摘要

Page Speed 还为 Firebug 添加了一个 Page Speed Activity 面板。该面板类似 Firebug 自带的网络面板，但它为每个请求提供了更加精确的数据。其中包括每个脚本生命周期的统计分析——包括解析和运行阶段，并提供了脚本之间时间线间隙的详细说明。这有助于分析特定区域，并在需要时重构这些文件。图 10-13 以红色标记解析脚本的时间，以蓝色标记脚本运行的时间。运行时间太长的脚本可能需要我们用分析工具深入研究。

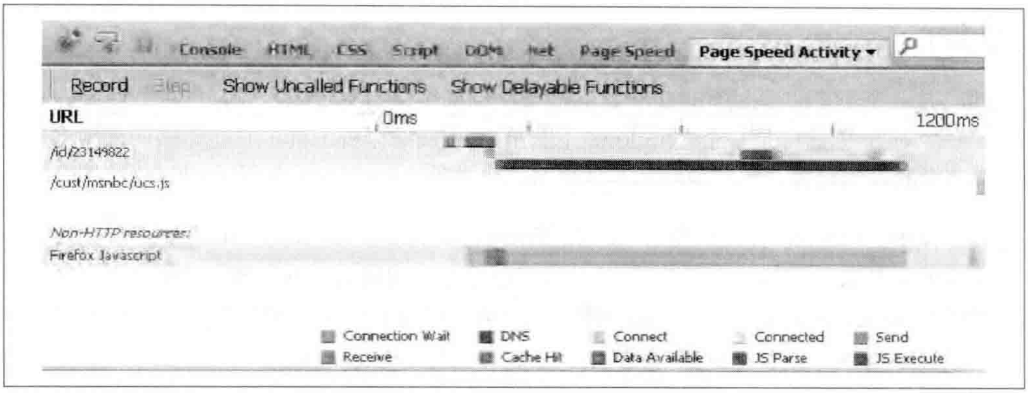
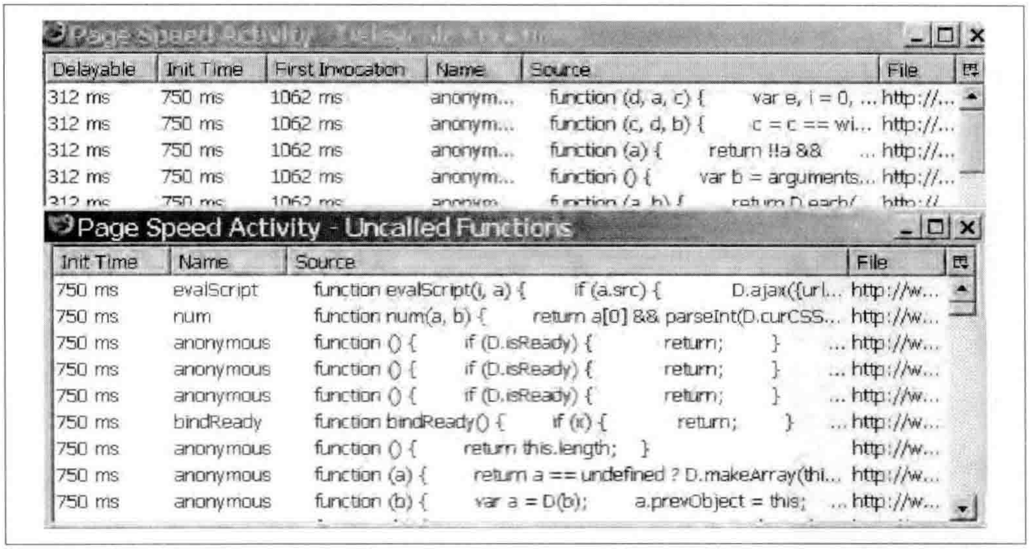

Console
HTML
CSS
Script
DOM
Net
Page Speed
Page Speed Activity
Record
Stop
Show Uncalled Functions
Show Delayable Functions
URL
0ms
1200ms
/id/23149822
/cust/msnbc/ucs.js
Non-HTTP resources:
Firefox Javascript
Connection Wait
DNS
Connect
Connected
Send
Receive
Cache Hit
Data Available
JS Parse
JS Execute

图 10-13　Page Speed 显示脚本的解析时间和运行时间

解析和初始化脚本可能花费了大量时间，而这些脚本在页面完成渲染后还未用到。Page Speed Activity 面板能提供报告指出哪些函数从未被调用，以及哪些函数可以延迟调用。结论基于它们的解析时间以及它们第一次被调用的时间（参见图 10-14）。



Delayable	Init Time	First Invocation	Name	Source	File
312 ms	750 ms	1062 ms	anonym...	function (d, a, c) { var e, i = 0, ...	http://...
312 ms	750 ms	1062 ms	anonym...	function (c, d, b) { c = c == wi...	http://...
312 ms	750 ms	1062 ms	anonym...	function (a) { return !!a && ...	http://...
312 ms	750 ms	1062 ms	anonym...	function () { var b = arguments...	http://...

Page Speed Activity - Uncalled Functions

Init Time	Name	Source	File
750 ms	evalScript	function evalScript(i, a) { if (a.src) { D.ajax({url...	http://w...
750 ms	num	function num(a, b) { return a[0] && parseInt(D.curCSS...	http://w...
750 ms	anonymous	function () { if (D.isReady) { return; } ...	http://w...
750 ms	anonymous	function () { if (D.isReady) { return; } ...	http://w...
750 ms	anonymous	function () { if (D.isReady) { return; } ...	http://w...
750 ms	bindReady	function bindReady() { if (x) { return; } ...	http://w...
750 ms	anonymous	function () { return this.length; }	http://w...
750 ms	anonymous	function (a) { return a == undefined ? D.makeArray(thi...	http://w...
750 ms	anonymous	function (b) { var a = D(b); a.prevObject = this; ...	http://w...

图 10-14　关于可延迟函数和未调用函数的报告

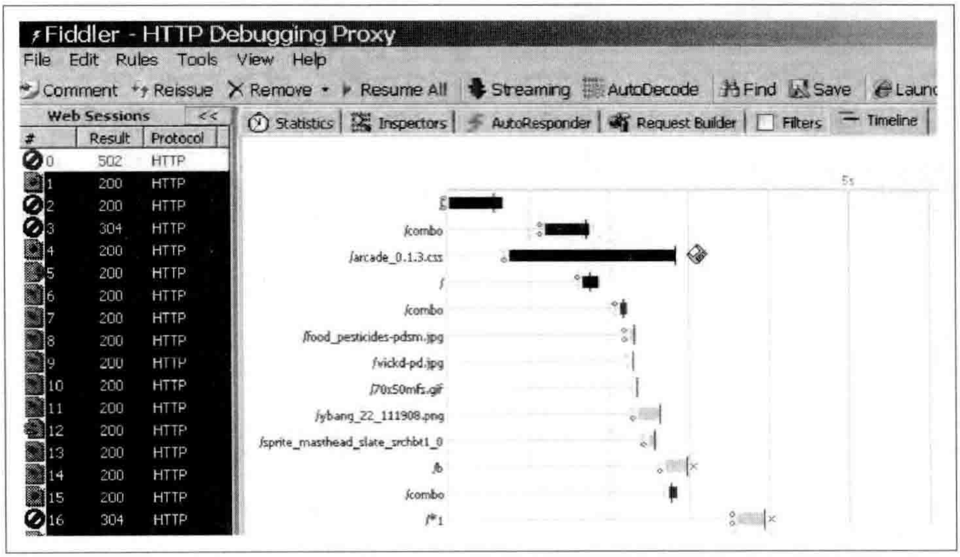
这些报告显示了那些从未被调用或随后才会被调用的函数初始化所用的总时间。你可以考虑重构代码，删除那些从未被调用的函数，并且延迟加载那些在初始渲染和设置阶段用不到的代码。

Fiddler

Fiddler 是一个 HTTP 调试代理工具，它能监测到网络中所有资源，以定位加载瓶颈。它由 Eric Lawrence 创建，是一个 Windows 下[译注8]的通用网络分析工具，它能提供任意浏览器或网络请求的详细报告。安装说明和其他更多信息参见 *http://www.fiddler2.com/fiddler2/*。

安装过程中，Fiddler 会自动整合 IE 和 Firefox。它会给 IE 工具栏中添加一个按钮，给 Firefox 的工具菜单添加一个菜单项。Fiddler 也可以手动启动。所有浏览器和应用程序产生的网络请求都会被分析。当 Fiddler 运行时，所有的 WinINE[译注9]通信将经由 Fiddler 中转，使得它可以监控并分析下载资源的性能。一些浏览器（比如 Opera 和 Safari）并不使用 WinINET，但是它们会自动检测 Fiddler 代理，如果 Fiddler 在这些浏览器启动前已运行的话。任何允许设置代理的程序都可以用 Fiddler 监控，只要把代理设置指向 Fiddler（127.0.0.1，端口：8888）。

像 Firebug、Web Inspector 和 Page Speed 一样，Fiddler 也提供了一个瀑布图，可以深入查看哪些资源文件占用了更长的下载时间，以及哪些资源文件可能会影响其他资源的加载（参见图 10-15）。


Fiddler - HTTP Debugging Proxy
File Edit Rules Tools View Help
Comment Reissue Remove Resume All Streaming AutoDecode Find Save Launc
Web Sessions <<
Statistics Inspectors AutoResponder Request Builder Filters Timeline
5s
/
/combo
/arcade_0.1.3.css
/
/combo
/food_pesticides-pdsm.jpg
/vickd-pd.jpg
/70x50mfs.gif
/ybang_22_111908.png
/sprite_masthead_slate_srchbt1_0
/b
/combo
/*1

#	Result	Protocol
0	502	HTTP
1	200	HTTP
2	200	HTTP
3	304	HTTP
4	200	HTTP
5	200	HTTP
6	200	HTTP
7	200	HTTP
8	200	HTTP
9	200	HTTP
10	200	HTTP
11	200	HTTP
12	200	HTTP
13	200	HTTP
14	200	HTTP
15	200	HTTP
16	304	HTTP

图 10-15 Fiddler 瀑布图

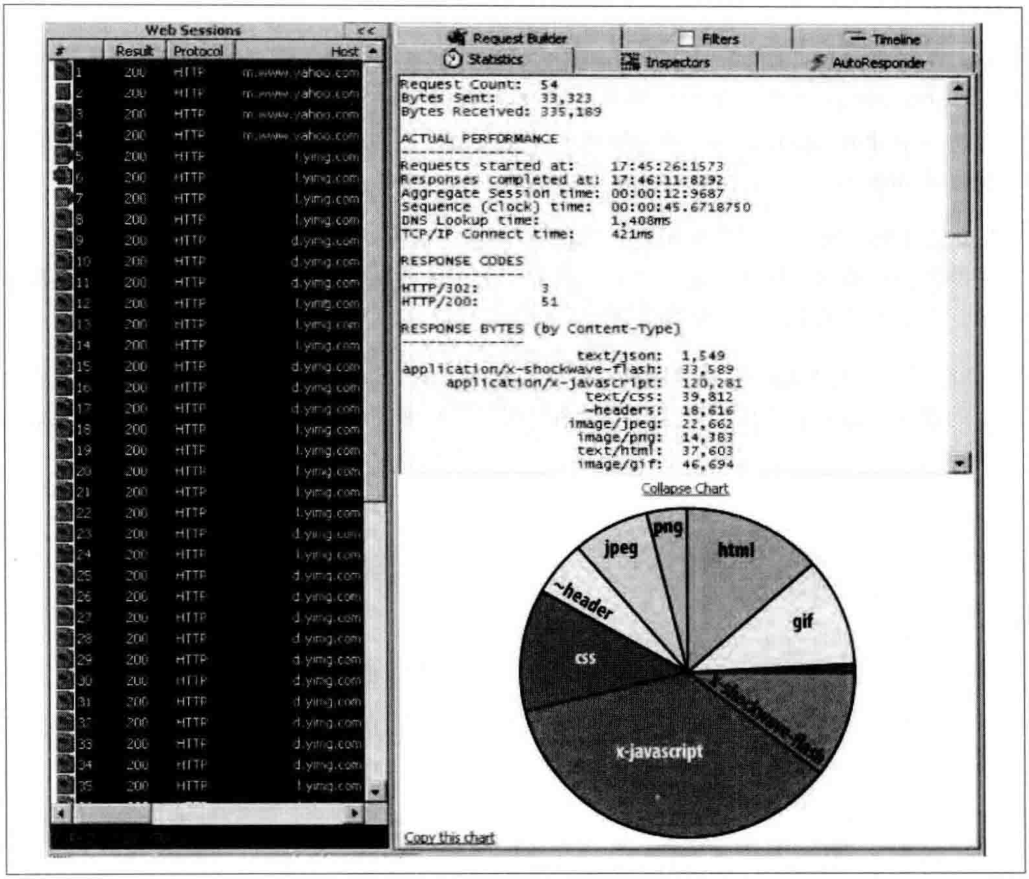
在左侧面板中选择一个或多个资源文件，点击 Timeline 标签就会在右侧主窗口中显示它们

[译注8] 在 OS X 系统下，首选推荐 Charles，详见：http://www.charlesproxy.com。

[译注9] Windows Internet（WinInet）是 Microsoft 提供的应用程序编程接口（API），它让开发人员可以方便地使用 HTTP、FTP 协议访问 Internet。更多信息请参见微软官方文档（英文）：*http://msdn.microsoft.com/en-us/ library/aa383630(VS.85).aspx*。

的瀑布图。这个视图提供了每个资源文件和与它关联的其他资源文件计时信息，方便你研究不同加载策略的效果，使得阻塞的状态更加明显地展现出来。

Statistics标签详细显示所选资源的实际性能状况除了可以深入查看DNS解析和TCP/IP连接时间，还包括所请求的各种资源的大小、类型等信息的明细图表（参见图10-16）。



#	Result	Protocol	Host
1	200	HTTP	m.www.yahoo.com
2	200	HTTP	m.www.yahoo.com
3	200	HTTP	m.www.yahoo.com
4	200	HTTP	m.www.yahoo.com
5	200	HTTP	l.yimg.com
6	200	HTTP	l.yimg.com
7	200	HTTP	l.yimg.com
8	200	HTTP	l.yimg.com
9	200	HTTP	d.yimg.com
10	200	HTTP	d.yimg.com
11	200	HTTP	d.yimg.com
12	200	HTTP	l.yimg.com
13	200	HTTP	l.yimg.com
14	200	HTTP	l.yimg.com
15	200	HTTP	d.yimg.com
16	200	HTTP	d.yimg.com
17	200	HTTP	d.yimg.com
18	200	HTTP	l.yimg.com
19	200	HTTP	l.yimg.com
20	200	HTTP	l.yimg.com
21	200	HTTP	l.yimg.com
22	200	HTTP	l.yimg.com
23	200	HTTP	d.yimg.com
24	200	HTTP	l.yimg.com
25	200	HTTP	d.yimg.com
26	200	HTTP	d.yimg.com
27	200	HTTP	d.yimg.com
28	200	HTTP	d.yimg.com
29	200	HTTP	d.yimg.com
30	200	HTTP	d.yimg.com
31	200	HTTP	d.yimg.com
32	200	HTTP	d.yimg.com
33	200	HTTP	d.yimg.com
34	200	HTTP	d.yimg.com
35	200	HTTP	l.yimg.com

```
Request Count:  54
Bytes Sent:     33,323
Bytes Received: 335,189

ACTUAL PERFORMANCE
--------------
Requests started at:      17:45:26:1573
Responses completed at:   17:46:11:8292
Aggregate Session time:   00:00:12:9687
Sequence (clock) time:    00:00:45.6718750
DNS Lookup time:          1,408ms
TCP/IP Connect time:      421ms

RESPONSE CODES
--------------
HTTP/302:       3
HTTP/200:       51

RESPONSE BYTES (by Content-Type)
--------------
                       text/json:  1,549
application/x-shockwave-flash:  33,589
     application/x-javascript:  120,281
                        text/css:  39,812
                       ~headers:  18,616
                     image/jpeg:  22,662
                      image/png:  14,383
                      text/html:  37,603
                      image/gif:  46,694
```

图 10-16 Fiddler 的 Statistics 标签

这些数据能帮助你决定哪些地方应该进行更深入的调查。例如，长时间的DNS解析和TCP/IP连接可能意味着网络问题。从资源饼图可以明显看出哪种类型的资源在页面加载中占有较大的比例，找出那些可以延迟加载的资源或者作进一步分析（如果是脚本类型的话）。

提示：Fiddler 只有 Windows 版本，值得一提的是，有一款名为 Charles Proxy 的共享软件可以同时在 Windows 和 Mac 下工作。访问 *http://www.charlesproxy.com* 获得试用版及详细文档。

YSlow

YSlow 工具可以深入观察页面初始加载和运行过程的整体性能。该工具最初由 Yahoo!内部的 Steve Sounders 开发，作为一个 Firefox 的插件（通过 GreaseMonkey 使用）。如今它已发布为一个 Firebug 的插件，由 Yahoo! 开发人员维护并定期更新。安装说明及其他产品信息参见 *http://developer.yahoo.com/yslow/*。

YSlow 给页面中加载的外部资源评分，提供页面性能报告，并给出提升加载速度的建议。它采用的评分标准基于性能专家们的大量研究。运用这些反馈信息，并阅读评分背后更多的细节，有助于以最少的资源提供最快的页面加载体验。

图 10-17 显示的是 YSlow 对网页进行分析后的默认视图。它将提供关于优化下载和页面渲染速度的建议。每一个评分都包含一个细节视图显示更多的附加信息，以及对规则基本原理的解释。

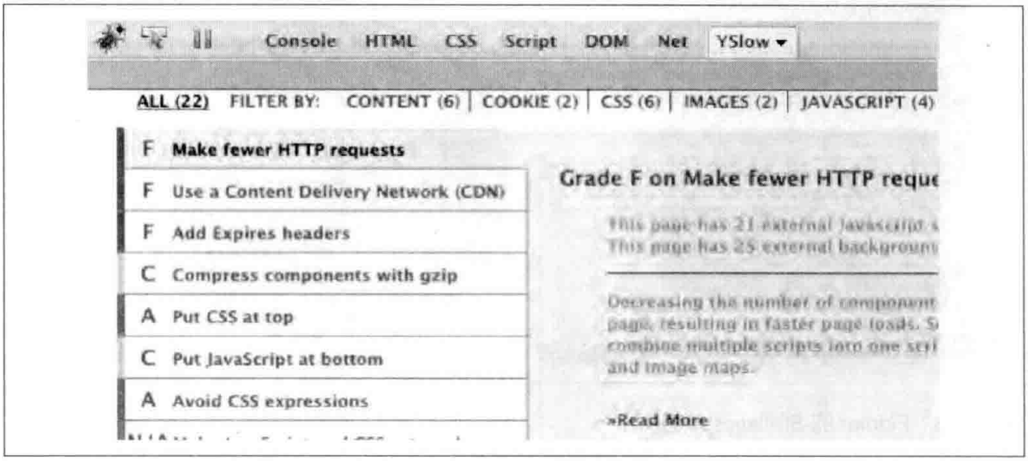
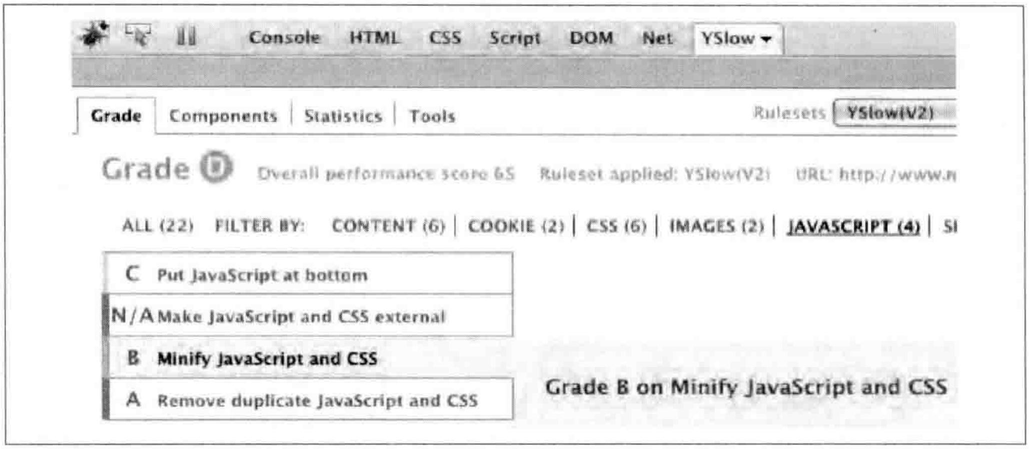

Console
HTML
CSS
Script
DOM
Net
YSlow ▾
ALL (22)
FILTER BY:
CONTENT (6)
COOKIE (2)
CSS (6)
IMAGES (2)
JAVASCRIPT (4)
F Make fewer HTTP requests
F Use a Content Delivery Network (CDN)
F Add Expires headers
C Compress components with gzip
A Put CSS at top
C Put JavaScript at bottom
A Avoid CSS expressions
Grade F on Make fewer HTTP reque
»Read More

图 10-17 YSlow：所有结果

一般来说，提升整体评分会导致页面的加载和脚本的执行变得更快。图 10-18 显示了 JAVASCRIPT 选项筛选后的结果，还有一些关于如何优化脚本传输和运行的建议。



Grade D Overall performance score 65 Ruleset applied: YSlow(V2) URL: http://www.n

ALL (22) FILTER BY: CONTENT (6) | COOKIE (2) | CSS (6) | IMAGES (2) | JAVASCRIPT (4) | SI

C Put JavaScript at bottom
N/A Make JavaScript and CSS external
B Minify JavaScript and CSS
A Remove duplicate JavaScript and CSS

Grade B on Minify JavaScript and CSS

图 10-18 YSlow：结果中 JavaScript 部分

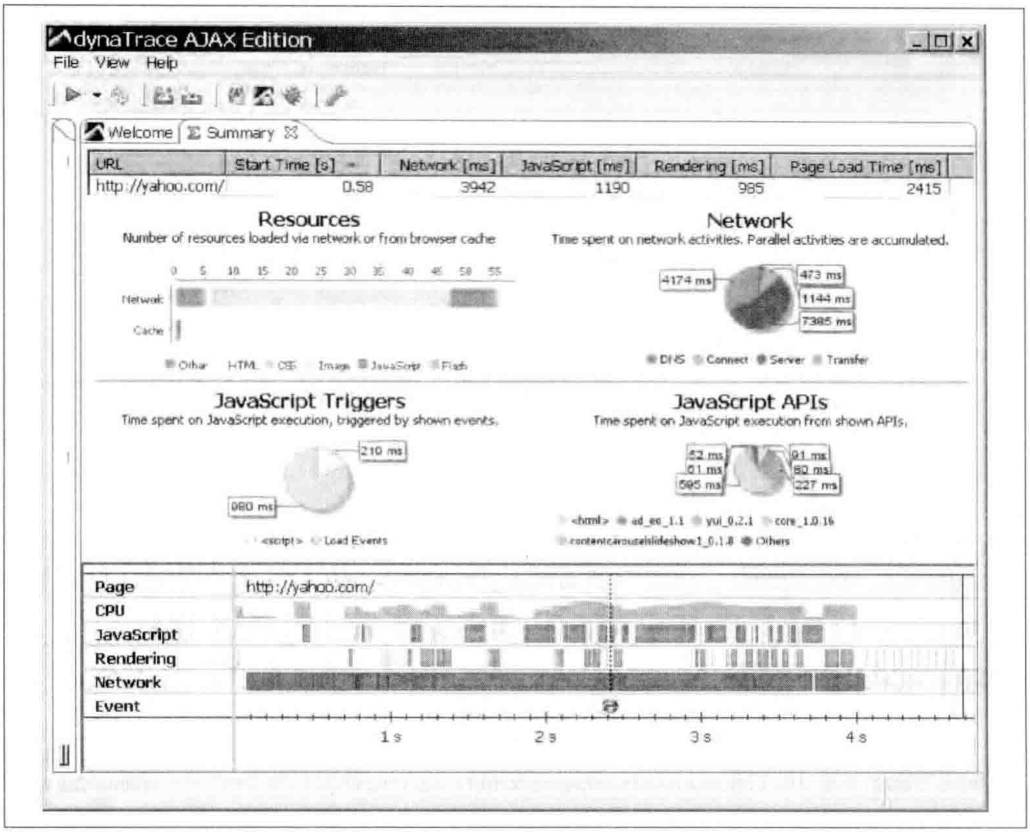
在分析结果时，请记住要考虑意外情况，包括决定是否将多个脚本请求合并成一个单独请求，以及哪些脚本或函数应当推迟到页面渲染之后加载。

dynaTrace Ajax Edition

dynaTrace 是一个强大的 Java/.NET 性能诊断工具，它的开发人员已经发布了一个“Ajax Edition”用来测量 IE 性能（Firefox 版即将发布）。这个免费工具提供了一个全面的性能分析，从网络和页面渲染到运行期脚本和 CPU 使用率。分析报告将所有信息汇总显示，因此你可以很容易地发现瓶颈所在。结果还可以导出用作进一步分析。它的下载地址是 *http://ajax.dynatrace.com/pages/*。

总结报告如图 10-19 所示，它提供了一个图形化的概况，使你可以很快发现哪些区域需要更多关注。从这里你可以深入到各种具体的报告中，更细粒度地查看特定方面的性能的细节。

如图 10-20 所示的网络视图，提供了关于网络生命周期的每个阶段耗费时间的详细报告，包括 DNS 查询、连接和服务器响应时间。它会指引你找到网络中可能需要调整的特定区域。报告下面的面板显示了请求和响应的头信息（左侧）以及实际的请求响应（右侧）。

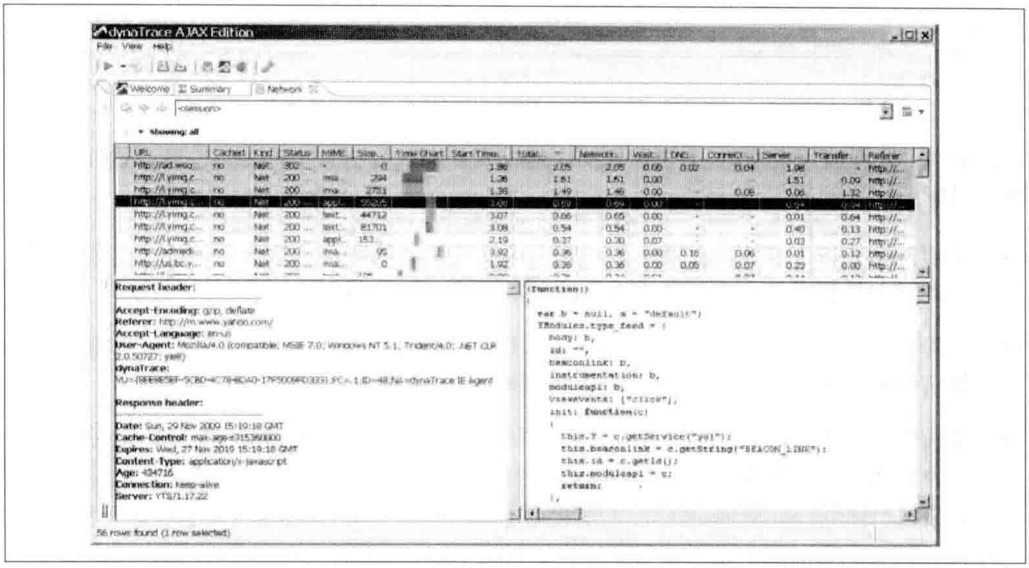

dynaTrace AJAX Edition
File
View
Help
Welcome
Summary
Resources
Number of resources loaded via network or from browser cache
Network
Cache
Other
HTML
CSS
Image
JavaScript
Flash
Network
Time spent on network activities. Parallel activities are accumulated.
4174 ms
473 ms
1144 ms
7385 ms
DNS
Connect
Server
Transfer
JavaScript Triggers
Time spent on JavaScript execution, triggered by shown events.
210 ms
980 ms
<script>
Load Events
JavaScript APIs
Time spent on JavaScript execution from shown APIs.
52 ms
61 ms
585 ms
91 ms
80 ms
227 ms
<html>
ad_eo_1.1
yui_0.2.1
core_1.0.16
contentcarouselslideshow1_0.1.8
Others
Page
http://yahoo.com/
CPU
JavaScript
Rendering
Network
Event
1 s
2 s
3 s
4 s

URL	Start Time [s]	Network [ms]	JavaScript [ms]	Rendering [ms]	Page Load Time [ms]
http://yahoo.com/	0.58	3942	1190	985	2415

图 10-19 dynaTrace Ajax Edition：汇总报告

选择 JavaScript 的 Triggers 视图将看到跟踪过程中被触发的每个事件的详细报告（参见图 10-21）。由此你可以深入到特定的事件（“load”、“click”、“mouseover” 等等）去发现运行期性能问题的根本原因。

这个视图中包括一个事件可能触发的任何动态（Ajax）请求，以及可能作为请求结果执行的任意脚本“回调”。这让你可以更好地理解用户所体会到的整体性能，由于 Ajax 的异步特性，它在脚本分析报告中可能不那么明显。



URL	Cached	Kind	Status	MIME	Size	Time Chart	Start Time	Total	Network	Wait	DNS	Connect	Server	Transfer	Referer
http://ad.wsq...	no	Net	302 ...	-	0		1.96	2.05	2.05	0.00	0.02	0.04	1.98	-	http://...
http://l.yimg.c...	no	Net	200	ima...	294		1.36	1.61	1.61	0.00			1.51	0.09	http://...
http://l.yimg.c...	no	Net	200	ima...	2751		1.38	1.49	1.46	0.00	-	0.06	0.06	1.32	http://...
http://l.yimg.c...	no	Net	200 ...	appl...	55205		3.00	0.59	0.59	0.00	-		0.54	0.04	http://...
http://l.yimg.c...	no	Net	200 ...	text...	44712		3.07	0.66	0.65	0.00	-	-	0.01	0.64	http://...
http://l.yimg.c...	no	Net	200 ...	text...	81701		3.08	0.54	0.54	0.00	-	-	0.40	0.13	http://...
http://l.yimg.c...	no	Net	200 ...	appl...	153		2.19	0.37	0.30	0.07	-	-	0.03	0.27	http://...
http://admedi...	no	Net	200 ...	ima...	95		3.92	0.36	0.36	0.00	0.16	0.06	0.01	0.12	http://...
http://us.bc.y...	no	Net	200 ...	ima...	0		1.92	0.36	0.36	0.00	0.05	0.07	0.23	0.00	http://...

Request header:

Accept-Encoding: gzip, deflate
Referer: http://m.www.yahoo.com/
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; yie8)
dynaTrace:
VU={BEEBE5EF-5CBD-4C78-8DA0-17F5009FD333};PC=.1;ID=48;NA=dynaTrace IE Agent

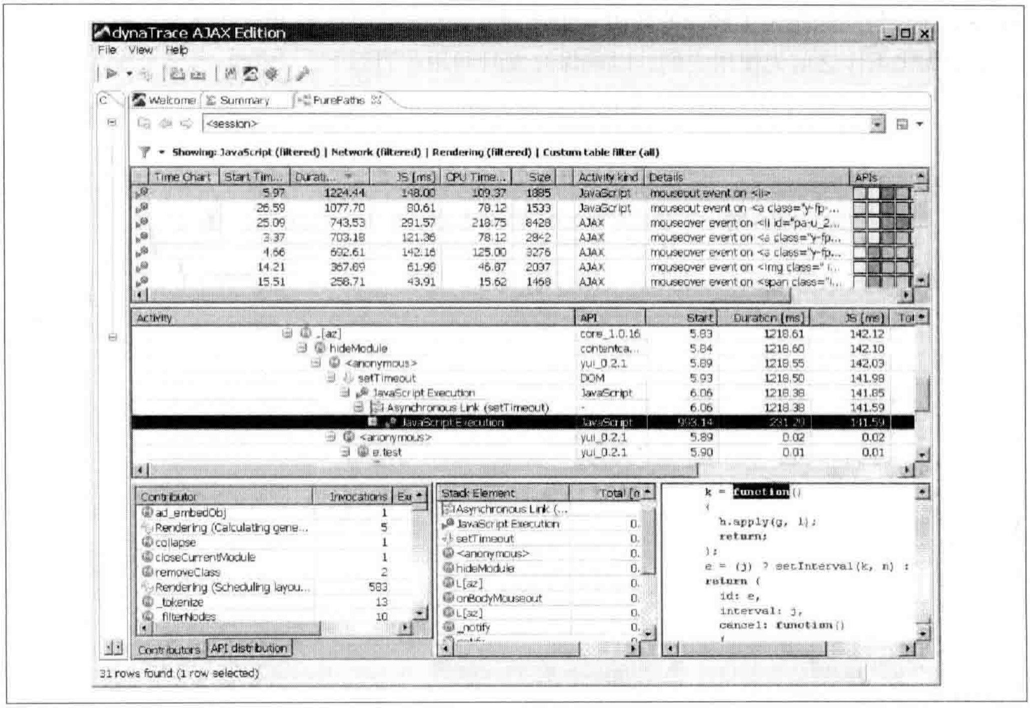
Response header:

Date: Sun, 29 Nov 2009 15:19:18 GMT
Cache-Control: max-age=315360000
Expires: Wed, 27 Nov 2019 15:19:18 GMT
Content-Type: application/x-javascript
Age: 434716
Connection: keep-alive
Server: YTS/1.17.22

```
(function()
{
  var b = null, a = "default";
  YModules.type_feed = {
    body: b,
    id: "",
    beaconlink: b,
    instrumentation: b,
    moduleapi: b,
    viewevents: ["click"],
    init: function(c)
    {
      this.Y = c.getService("yui");
      this.beaconlink = c.getString("BEACON_LINK");
      this.id = c.getId();
      this.moduleapi = c;
      return;
    },
```

56 rows found (1 row selected)

图 10-20　dynaTrace Ajax Edition：网络报告



Showing: JavaScript (filtered) | Network (filtered) | Rendering (filtered) | Custom table filter (all)

Time Chart	Start Tim...	Durati...	JS [ms]	CPU Time...	Size	Activity kind	Details
	5.97	1224.44	148.00	109.37	1885	JavaScript	mouseout event on
	26.59	1077.70	80.61	78.12	1533	JavaScript	mouseout event on <a class="y-fp-...
	25.09	743.53	291.57	218.75	8428	AJAX	mouseover event on <li id="pa-u_2...
	3.37	703.18	121.36	78.12	2842	AJAX	mouseover event on <a class="y-fp...
	4.66	692.61	142.16	125.00	3276	AJAX	mouseover event on <a class="y-fp...
	14.21	367.69	61.90	46.87	2037	AJAX	mouseover event on <img class=" i...
	15.51	258.71	43.91	15.62	1468	AJAX	mouseover event on <span class="i...

Activity	API	Start	Duration [ms]	JS [ms]
_[az]	core_1.0.16	5.83	1218.61	142.12
hideModule	contentca...	5.84	1218.60	142.10
<anonymous>	yui_0.2.1	5.89	1218.55	142.03
setTimeout	DOM	5.93	1218.50	141.98
JavaScript Execution	JavaScript	6.06	1218.38	141.85
Asynchronous Link (setTimeout)	-	6.06	1218.38	141.59
JavaScript Execution	JavaScript	993.14	231.20	141.59
<anonymous>	yui_0.2.1	5.89	0.02	0.02
e.test	yui_0.2.1	5.90	0.01	0.01

Contributor	Invocations
ad_embedObj	1
Rendering (Calculating gene...	5
collapse	1
closeCurrentModule	1
removeClass	2
Rendering (Scheduling layou...	583
_tokenize	13
filterNodes	10

Stack Element	Total [n
Asynchronous Link (...	
JavaScript Execution	0.
setTimeout	0.
<anonymous>	0.
hideModule	0.
L[az]	0.
onBodyMouseout	0.
L[az]	0.
_notify	0.

```
k = function()
{
  h.apply(g, l);
  return;
};
e = (j) ? setInterval(k, n) :
return {
  id: e,
  interval: j,
  cancel: function()
  {
```

31 rows found (1 row selected)

图 10-21　dynaTrace Ajax Edition Pure Paths 面板

小结

Summary

当网页或 Web 应用变慢时，分析从网络下载的资源以及分析脚本的运行性能能让你专注于那些最需要优化的地方。

- 使用网络分析工具找出加载脚本和页面中其他资源的瓶颈，这会帮助你决定哪些脚本需要延迟加载，或者需要进一步分析。
- 尽管传统的经验告诉我们要尽量减少 HTTP 请求数，但把脚本尽可能延迟加载可以加快页面渲染速度，给用户带来更好的体验。
- 使用性能分析工具找出脚本运行过程中速度慢的地方，检查每个函数所消耗的时间，以及函数被调用的次数，通过调用栈自身提供的一些线索来找出需要集中精力优化的地方。
- 尽管耗费的时间和调用次数通常是数据中最有价值的部分，但仔细观察函数的调用过程，你也许会发现其他优化目标。

这些工具会帮助你深入了解你的代码在那些通常你比较陌生的编程环境下是如何运行的。在开始优化工作之前先使用它们，以确保开发时间用在刀刃上。

索引
Index

符号

+ (plus) operator（加操作符）, 82–84
+= (plus-equals) operator（加等操作符）, 82–84

A

activation objects（活动对象）, 18
add() function（add() 函数）, 17
addEventListener() function（addEventListener() 函数）, 154
Agile JavaScript build process（敏捷 JavaScript 构建过程）, 174
Ajax (Asynchronous JavaScript and XML)（异步 JavaScript 和 XML）, 125–150
    data formats（数据格式）, 134–145
    data transmission（数据传输）, 125–134
    performance（性能）, 145–149
algorithms（算法）, 61–80
    conditionals（条件语句）, 68–73
    loops（循环）, 61–68
    recursion（迭代）, 73–79
alternation（分支）
    and backtracking（与回溯）, 89
    performance（性能）, 96
animations, elements and flow（动画、元素和文档流）, 56
anonymous functions（匿名函数）
    about（关于）, 182
    YUI Profiler, 181
Apache Ant, 163, 173
Apache web server, ExpiresDefault directive（Apache Web 服务器, ExpiresDefault 指令）,172
APIs
    console API: Firebug（控制台 API：Firebug）, 184
    CSS selectors（CSS 选择器）, 48
    DOM, 35
array items, defined（数组元素，定义）, 15
array joining（数组项连接）, 84
array processing, with timers（数组处理，使用定时器）, 114
array.join() method（array.join 方法）, 82
arrays（数组）
    compared to HTMLCollection objects（对比 HTMLCollection 对象）, 42
    double evaluation（双重求值）, 153
    for-in loops（for-in 循环）, 63
assignEvents() function（ assignEvents() 函数）, 25
Asynchronous JavaScript and XML (see Ajax)（参见“Ajax”）
atomic grouping, emulating（原子组，模拟）, 93

B

backreferences, emulating atomic groups（反向引用，模拟原子组）, 93
backtracking（回溯）, 89–95
    about（关于）, 89
    runaway backtracking（回溯失控）, 91–95
batching DOM changes（批量修改 DOM）, 54
beacons, sending data（信标，发送数据）, 133
benchmarking, regular expressions（基准测试，正则表达式）, 96
best practices (see programming practices)（参见“编程实践”）
bitmasking（位掩码）, 158
bitwise operators（位运算符）, 156–159
blocking scripts（阻塞脚本）, 193
BOM (Browser Object Model)（浏览器对象模型）, object members（对象成员）, 27
bracket notation versus dot notation（括号表示法对比点表示法）, 31

欢迎大家提出建议来改善索引内容。建议请发送 email 到 *index@oreilly.com*。

browsers（浏览器）, 1
  (参见 Chrome; Firefox; IE; Safari)
  call stack limits（调用栈限制）, 74
  DOM and JavaScript implementations（DOM 和 JavaScript 实现）, 35
  DOM scripting（DOM 编程）, 35
  native code versus eval()（原生代码对比 eval()）, 152
  performance（性能）, 1, 15
  script blocking（脚本块）, 193
  trim, 103
  UI threads（UI 线程）, 107
  WebKit-based and innerHTML（基于 Web-kit 和 innerHTML）, 38
  XPath, 137
build process, Agile JavaScript（构建过程，敏捷 JavaScript）, 174
buildtime versus runtime build processes（构建时处理对比运行时处理）, 170

C

caching（缓存）
  Ajax, 145
  JavaScript files（JavaScript 文件）, 171
  layout information（布局信息）, 56
  object member values（对象成员值）, 31
  using（使用）, 172
call stack size limits（调用栈大小限制）, 74
CDN (content delivery network)（内容分发网络）, 173
chains (see prototype chains; scope chains)（参见“原型链；所用域链”）
childNodes collection（childNodes 集合）, 47
Chrome
  developer tools（开发人员工具）, 192
  just-in-time JavaScript compiler（实时 JavaScript 编译器）, 160
  time limits（时间限制）, 110
cloning nodes（克隆节点）, 41
Closure Compiler, 169
Closure Inspector, 169
closures, scope（闭包，作用域）, 24
collections（集合）
  childNodes collection（childNodes 集合）, 47
  collection elements（集合元素）, 45
  HTML collections（HTML 集合）, 42–46
combining JavaScript files（合并 JavaScript 文件）, 165
compile-time folding, Firefox（编译期合并，Firefox）, 84
compression（压缩）, 170
concat method（concat 方法）, 86
concatenating strings（连接字符串）, 40, 81–87
conditional advance loading（条件预加载）, 156
conditionals（条件语句）, 68–73
  if-else, 68, 70
  lookup tables（查找表）, 72
console API, Firebug（控制台 API, Firebug）, 184
Console panel profiler, Firebug（控制台面板分析工具，Firebug）, 183
console.time() function（console.time() 函数）, 185
constants, mathematical constants（常量，数学常量）, 159
content delivery network (CDN), 173
crawling DOM（遍历同级节点）, 47
CSS files, loading（CSS 文件，加载）, 13
CSS selectors, APIs（CSS 选择器，APIs）, 48
cssText property（cssText 属性）, 53

D

data access（数据存取）, 15–33
  object members（对象成员）, 27–33
  scope（作用域）, 16–26
data caching（数据缓存）, 145
data formats（数据格式）, 134–145
  custom（自定义）, 142
  HTML, 141
  JSON, 137–141
  XML, 134–137
data transmission（数据传输）, 125–134
  requesting data（请求数据）, 125–131
  sending data（发送数据）, 131–134
data types: functions, methods and properties（数据类型：函数、方法和属性）,27
Date object（数据对象）, 178
deferred scripts（延迟脚本）, 5
delegation（委托）, events（事件）, 57
deploying JavaScript resources（部署 JavaScript 资源）, 173
displayName property（displayName 属性）, 190
do-while loops（do-while 循环）, 62
document fragments, batching DOM changes（文档片段，批量修改 DOM）, 55
DOM (Document Object Model)（文档对象模型）, object members（对象成员）, 27
DOM scripting（DOM 编程）, 35–59
  access document structure（访问文档结构）, 46–50
  browsers（浏览器）, 35
  cloning nodes（克隆节点）, 41
  event delegation（事件委托）, 57
  HTML collections（HTML 集合）, 42–46
  innerHTML, 37–40

  repaints and reflows（重绘和重排）, 50–57
dot notation versus bracket notation（点表示法对比括号表示法）, 31
double evaluation（双重求值）, 151–153
downloading（下载）, 122
  (see also DOM scripting; loading; nonblocking scripts; scripts)（参见“DOM 编程；加载；无阻塞脚本；脚本”）
  blocking by <script> tags（被 <script> 标签阻塞）, 3
  using dynamic script nodes（使用动态脚本节点）, 7
dynamic scopes（动态作用域）, 24
dynamic script elements（动态脚本元素）, 6–9
dynamic script tag insertion（动态脚本注入）, 127
dynaTrace, 199

E

element nodes（元素节点）, DOM, 47
elements（元素）, 45
  (see also collections; <script> elements;tags))（参见“集合”；<script> 元素；标签）
  DOM, 50
  dynamic script elements（动态脚本元素）, 6–9
  reflows（重排）, 56
emulating atomic groups（模拟原子组）, 93
eval() function（eval() 函数）, 24, 138, 151
event delegation（事件委托）, DOM scripting（DOM 编程）, 57
events（事件）
  message events（消息事件）, 121
  onmessage events（onmessage 事件）, 121
  readystatechange events（readystatechange 事件）, 7
execute() function（execute() 函数）, 24
execution (see scripts)（参见“脚本”）
Expires headers（Expires 头信息）, 146
ExpiresDefault directive（ExpiresDefault 指令）, Apache web server（Apache Web 服务器）,172
external files（外部文件）, loading（加载）, 122

F

factorial() function（factorial() 函数）, 75, 78
Fiddler, 196
files（文件）, 122
  (see also DOM scripting; downloading;loading; nonblocking scripts; scripts)（参见“DOM 编程；下载；加载；无阻塞脚本；脚本”）
  caching JavaScript files（缓存 JavaScript 文件）, 171
  combining JavaScript files（合并 JavaScript 文件）, 165
  loading external files（加载外部文件）, 122
  preprocessing JavaScript files（预处理 JavaScript 文件）, 166
Firebug, 183–186
Firefox
  compile-time folding（编译器合并）, 84
  time limits（时间限制）, 110
flow control（流程控制）, 61–80
  conditionals（条件语句）, 68–73
  loops（循环）, 61–68
  recursion（迭代）, 73–79
flows (see reflows)（参见“重排”）
flushing render tree changes（刷新渲染树变化）, 51
folding, compile-time folding and Firefox（合并，编译期合并和 Firefox）, 84
for loops（for 循环）, 62
for-in loops（for-in 循环）, 62, 63
forEach() method（forEach() 方法）, 67
Function() constructor（Function() 构造函数）, 151
functions（函数）, 116
  (see also methods; statements)（参见“方法”；“语句”）
  add() function（add() 函数）, 17
  addEventListener() function（addEventListener() 函数）, 154
  anonymous functions（匿名函数）, 181, 182
  assignEvents() function（assignEvents() 函数）, 25
  caching object member values（缓存对象成员值）, 31
  console.time() function（console.time() 函数）, 185
  data types（数据类型）, 27
  eval() function（eval() 函数）, 24, 138, 151
  execute() function（execute() 函数）, 24
  factorial() function（factorial() 函数）, 75, 78
  initUI() function（initUI() 函数）, 21
  loadScript() function（loadScript() 函数）, 11
  mergeSort() function（mergeSort() 函数）, 77
  multistep() function（multistep() 函数）, 118
  processArray() function（processArray() 函数）, 116

- profileEnd() function（profileEnd() 函数），184
- removeEventListener() function（removeEventListener() 函数），154
- setInterval() function （setInterval() 函数），112, 151
- setTimeout() function（setTimeout() 函数），112, 151
- tasks（任务），116
- variables in execution（运行期变量），18

G

GET versus POST when using XHR（使用 XHR 时，GET 与 POST 对比），127
global variables（全局变量）, performance（性能），19
Google Chrome developer tools（Google）（Google Chrome 开发人员工具），192
Google Closure Compiler, 169
grouping scripts（组织脚本），4
gzip compression（Gzip 压缩），169, 170

H

handleClick() method（handleClick() 方法），108
hasOwnProperty() method（hasOwnProperty() 方法），28
headers（头信息）
- Expires headers（Expires 头信息），146
- HTTP headers（HTTP 头信息），146

:hover, IE, 57
HTML collections（HTML 集合）
- expensive collections（昂贵的集合），43
- local variables（局部变量），45

HTML, data format（数据格式），141
HTTP headers（HTTP 头信息），Ajax, 146

I

idempotent action（幂等行为），127
identifier resolution（标识符解析），scope（作用域），16–21
IE (Internet Explorer)
- array joining（数组项连接），84
- concatenating strings（连接字符串），40
- dynamic script elements（动态脚本元素），7
- nextSibling, 47
- reflows（重排），57
- repeated actions（重复任务），111
- time limits（时间限制），109
- using（使用），186
- XHR objects（XHR 对象），148

if-else
- optimizing（优化），70
- versus switch（对比 switch），68

initUI() function（initUI() 函数），21
injecting nonblocking scripts（注入无阻塞脚本），9
innerHTML
- data format（数据格式），141
- versus DOM（对比 DOM），37–40

interfaces (see user interfaces)（参见“用户界面”）
Internet Explorer (see IE)（参见“IE”）
interoperability（通用性），JSON, 140
iPhone (see Safari)（参见“Safari”）
iteration（迭代）
- function-based（基于函数的），67
- loop performance（循环性能），63–67
- recursion（递归），76

J

JavaScript files（JavaScript 文件）
- caching（缓存），171
- combining（合并），165
- preprocessing（预处理），166

JavaScript namespacing（JavaScript 命名空间），nested properties（嵌套属性），32
JavaScript profiling（JavaScript 分析），178
joining arrays（连接数组项），84
jQuery library（jQuery 类库），gzipping, 171
JSMin, 168
JSON (JavaScript Object Notation)（JavaScript 对象表示法）, data formats（数据格式），137–141
JSON-P (JSON with padding)（JSON 填充），139

L

$LAB.script() method（$LAB.script() 方法），13
$LAB.wait() method（$LAB.wait() 方法），13
LABjs library（LABjs 类库），loading JavaScript（加载 JavaScript），13
layouts（布局），caching（缓存），56
lazy loading（延迟加载），154
LazyLoad library（LazyLoad 类库），loading JavaScript（加载 JavaScript），12
length property（length 属性），43
libraries（类库）
- Ajax, 148
- LABjs library（LABjs 类库），13
- LazyLoad library（LazyLoad 类库），12

limits（限制）
- call stack size limits（调用栈大小限制），74
- long-running script limit（长运行脚本限制），109

literal values（直接量），defined（定义），15

loading（加载）, 122
  (see also DOM scripting; downloading; nonblocking scripts; scripts)(参见“DOM 编程”；“下载”；“无阻塞脚本”；“脚本”)
  conditional advance loading（条件预加载）, 156
  CSS files（CSS 文件）, 13
  external files（外部文件）, 122
  JavaScript, 10
  lazy loading（延迟加载）, 154
  scripts（脚本）, 192
loadScript() function（loadScript() 函数）, 11
local variables（局部变量）
  HTML collections（HTML 集合）, 45
  performance（性能）, 19, 36
long-running script limit（长运行脚本限制）, 109
lookaheads（向前查看）, emulating atomic groups（模拟原子组）, 93
lookup tables（查找表）, 72
loops（循环）, 61–68
  function-based iteration（函数迭代）, 67
  measuring timing with console.time()（用 console.time() 计时）, 185
  performance（性能）, 63–67
  types of, 61

M

mathematical constants and methods（数学常量和方法）, lists of,159
memoization, recursion（递归）, 77
mergeSort() function（mergeSort() 函数）, 77
message events（消息事件）, 121
methods（方法）, 159
  (see also functions; object members; properties; statements)(参见“函数”；“对象成员”；“属性”；“语句”)
  Array.prototype.join method (Array.prototype.join 方法）, 84
  concat method（concat 方法）, 86
  data types（数据类型）, 27
  forEach() method（forEach() 方法）, 67
  handleClick() method (handleClick() 方法), 108
  hasOwnProperty() method (hasOwnProperty() 方法）, 28
  $LAB.script() method ($LAB.script() 方法), 13
  $LAB.wait() method ($LAB.wait() 方法), 13
  mathematical methods（数学方法）, 159
  native methods（原生方法）, 159
  postMessage() method(postMessage() 方法), 121
  querySelector() method（querySelector() 方法）, 160
  querySelectorAll() method（querySelectorAll() 方法）, 48, 160
  string concatenation（字符串连接）, 82
  this in object methods（在对象方法中）, 33
  toString() method（toString() 方法）, 28
  trim method（trim 方法）, 99
minification（压缩）, 168
multistep() function（multistep() 函数）, 118
MXHR (multipart XHR), 128–131

N

namespacing（命名空间）, nested properties（嵌套属性）, 32
native methods（原生方法）, 159
nested object members（嵌套对象成员）, 30
nested quantifiers（嵌套量词）, runaway backtracking（回溯失控）, 94
Net panel（网络面板）, Firebug, 185
Nielsen, Jakob, on UI response time（UI 响应事件）, 110
nodes（节点）, cloning（克隆）, 41
nonblocking scripts（无阻塞脚本）, 5–14
  deferred scripts（延迟脚本）, 5
  dynamic script elements（动态脚本元素）, 6–9
  loading JavaScript（加载 JavaScript）, 10
  XMLHttpRequest Script Injections (XMLHttpRequest 脚本注入）, 9
noncapturing groups（非捕获组）, 97

O

object members（对象成员）, 27
  (see also methods; properties)(参见“方法”、“属性”)
  caching object member values（缓存对象成员值）, 31
  data access（数据访问）, 27–33
  defined（定义）, 15
  nested（嵌套）, 30
  prototype chains（原型链）, 29
  prototypes（原型）, 27
objects（对象）
  activation objects（活动对象）, 18
  Date object（Date 对象）, 178
  HTMLCollection, 42
  programming practices（编程实践）,153
  XHR objects（XHR 对象）, 148
onmessage events（onmessage 事件）, 121

- Opera, time limits（时间限制）, 110
- operators（操作符）
  - bitwise operators（位操作符）, 156–159
  - plus (+) and plus-equals(+=) operators（加操作符和加等操作符）, 82–84
- optimizing (see performance)(优化，参见“性能”)
- out-of-scope variables（跨作用域变量）, 26

P

- Page Speed, 194
- panels（面板）
  - Console panel profiler: Firebug（控制台面板分析工具：Firebug）, 183
  - Net panel: Firebug(网络面板：Firebug), 185
  - Profiles panel（分析面板）, 189
  - Resources panel: Safari Web Inspector（资源面板：Safari Web Inspector）,191
- parse times（解析时间）, XML, 137
- parsing(解析), eval() function with JSON（eval()函数和 JSON）, 138
- performance（性能）
  - Ajax, 145–149
  - array joining（数组项连接）, 84
  - browsers（浏览器）, 15
  - closures（闭包）, 25
  - DOM scripting（DOM 编程）, 35, 36
  - format comparison（格式比较）, 144
  - HTML format（HTML 格式）, 142
  - identifier resolution（标识符解析）, 19
  - JavaScript engines（JavaScript 引擎）, 24
  - JavaScript in browsers（浏览器中的 JavaScript）, 1
  - JSON formats（JSON 格式）, 139
  - JSON-P formats（JSON-P 格式）, 140
  - loops（循环）, 63–67
  - native code versus eval()（原生代码对比 eval()）, 152
  - nested members（嵌套成员）, 31
  - regexes（正则表达式）, 87, 96
  - timers（定时器）, 119
  - trim implementations（trim 实现）, 103
  - XHR formats（XHR 格式）, 144
  - XML, 137
- plus (+) operator（加操作符）, 82–84
- plus-equals (+=) operator（加等操作符）, 82–84
- positioning（定位）, scripts（脚本）, 2
- POST versus GET when using XHR（使用 XHR 时比较 POST 和 GET）, 127
- postMessage() method（postMessage() 方法）, 121
- preprocessing JavaScript files（预处理 JavaScript 脚本）, 166
- pretest conditions（前测条件）, loops（循环）, 62
- processArray() function（processArray() 函数）, 116
- profileEnd() function（profileEnd() 函数）, 184
- Profiler (YUI), 179–182
- Profiles panel（分析面板）, Safari Web Inspector, 189
- profiling（分析）, JavaScript, 178
- programming practices（编程实践）, 151–161
  - bitwise operators（位操作符）, 156–159
  - double evaluation（双重求值）, 151–153
  - native methods（原生方法）, 159
  - object/array literals（对象/数组直接量）, 153
  - repeating work（重复工作）, 154
- prop variable（prop 变量）, 62
- properties（属性）, 27
  - (see also methods; object members)(参见“方法”、“对象成员”)
  - cssText property（cssText 属性）, 53
  - data types（数据类型）, 27
  - displayName property（displayName 属性）, 190
  - DOM properties（DOM 属性）, 47
  - innerHTML property（innerHTML 属性）, 37
  - length property（length 属性）, 43
  - prototypes（原型）, 27
  - reading in functions（从函数读取）, 32
  - readyState properties (<script> element)（readyState 属性，<script> 元素）, 7
  - [[Scope]] property（[[Scope]] 属性）
- prototype chains（原型链）, object members（对象成员）, 29
- prototypes（原型）, object members（对象成员）, 27

Q

- quantifiers（量词）
  - nested quantifiers（嵌套量词）, 94
  - performance（性能）, 98
- queries（查询）, HTML collections（HTML 集合）, 43
- querySelector() method（querySelector() 方法）, 160
- querySelectorAll() method（querySelectorAll() 方法）, 48, 160

R

- readyState
  - MXHR, 130
  - XHR, 126

    XMLHttpRequest, 148
readyState properties (<script> element)（readyState 属性，<script> 元素），7
readystatechange events (readystatechange 事件), IE, 7
recursion（递归），73–79
    call stack size limits（调用栈限制），74
    iteration（迭代），76
    memoization, 77
    patterns（模式），75
reflows（重排），50–57
    caching layout information（缓存布局信息），56
    elements（元素），56
    IE, 57
    minimizing（最小化），52–56
    queuing and flushing render tree changes（渲染树变化的排队与刷新），51
regular expressions (regexes)（正则表达式），87–99
    about（关于），88
    atomic grouping（原子组），93
    backtracking（回溯），89, 91
    benchmarking（基准测试），96
    performance（性能），87, 96, 99
    repetition（重复），90
    trimming strings（去除字符），99, 100, 103
    when not to use（何时不使用），99
    removeEventListener() function（removeEventListener() 函数），154
    render trees DOM（渲染树 DOM），50
    reflows（重排），51
repaints（重绘），minimizing（最小化），52–56
repeating work（重复工作），154
repetition and backtracking（重复和回溯），90
requesting data（请求数据），Ajax, 125–131
Resources panel: Safari Web Inspector（资源面板：Safari Web Inspector），191
runaway backtracking（回溯失控），91
runtime build versus buildtime processes（运行时处理对比编译时处理），170

S

Safari
    caching ability（缓存能力），172
    loading scripts（加载脚本），192
    passing strings（传递参数），122
    starting and stopping profiling programmatically（自动启用或停止性能分析），189
    time limits（时间限制），110
Safari Web Inspector, 188–192
scope（作用域），16–26
    closures（闭包），24
    dynamic scopes（动态作用域），24
    identifier resolution（标识符解析），16–21
scope chains（作用域链）
    augmentation（改变），21
    identifier resolution（标识符解析），16
    performance（性能），20
[[Scope]] property（[[Scope]] 属性），25
script blocking（脚本阻塞），193
script tags（脚本标签），dynamic insertion（动态注入），127
<script> elements（<script> 元素）
    defer option（延迟选项），6
    DOM, 6
    performance（性能），1, 4
    placement of（替换），2
scripts（脚本），1–14
    (see also DOM scripting)（参见“DOM”编程）
    debugging and profiling（调试和性能分析），183
    grouping（分组），4
    loading（加载），192
    nonblocking scripts（无阻塞脚本），5–14
    positioning（定位），2
selectors（选择器），CSS, 48
sending data（发送数据），Ajax, 131–134
setInterval() function（setInterval() 函数），112, 151
setTimeout() function (setTimeout() 函数), 112, 151
smasher, 174
speed (see performance)（参见“性能”）
stacks（栈），call stack size limits（调用栈大小限制），74
statements（语句），116
    (see also conditionals; functions; methods)（参见“条件”、“函数”、“方法”）
    try-catch statements (try-catch 语句), 23, 75
    var statement（var 语句），62
    with statements（with 语句），21
string.concat() method (string.concat() 方法), 82
strings（字符串）
    concatenating（连接），40, 81–87
    passing in Safari（在 Safari 中传送），122
    trimming（去除），99–103
styles (样式), repaints and reflows (重绘和重排), 53
switches, if-else versus switch（if-else 对比 switch），68

T

- tables（表格）, lookup tables（查找表）, 72
- tags（标签）, 127
  - (see also elements)（参见"元素"）
  - <user> tags（<user> 标签）, 136
  - dynamic script tag insertion（动态脚本注入）, 127
- this, object methods（对象方法）, 33
- Thomas, Neil, on multiple repeating timers（多个重复定时器）,120
- threads（线程）, browser UI threads（浏览器 UI 线程）, 107
- time limits（时间限制）, browsers（浏览器）, 109–111
- timers（定时器）
  - performance（性能）, 119
  - yielding with（让出）, 111–120
- tokens（字元）, exposing（可见）, 98
- tools（工具）, 177–202
  - anonymous functions with（匿名函数）, 182
  - Chrome developer tools（Chrome 开发人员工具）, 192
  - dynaTrace, 199
  - Fiddler, 196
  - Firebug, 183–186
  - IE (Internet Explorer), 186
  - JavaScript profiling（JavaScript 性能分析）, 178
  - Page Speed, 194
  - Safari Web Inspector, 188–192
  - script blocking（脚本阻塞）, 193
  - YSlow, 198
  - YUI Profiler, 179–182
- toString() method（toString() 方法）, 28
- trees (see render trees)（树，参见"渲染树"）
- trimming strings（去除字符串）, 99–103, 99, 103
- try-catch statements（try-catch 语句）, 23, 75

U

- user interfaces（用户界面）, 107–124
  - browser UI threads（浏览器 UI 线程）, 107–111
  - web workers, 120–124
  - yielding with timers（让出时间片段）, 111–120
- <user> tags（<user> 标签）, 136

V

- values（值）, caching object member values（缓存对象成员值）, 31
- var statement（var 语句）, 62
- variables（变量）, 19
  - (see also local variables)（参见"局部变量"）
  - defined（定义）, 15
  - function execution（函数执行）, 18
  - local versus global（局部对比全局）, 19
  - out-of-scope variables（外部作用域变量）, 26
  - prop variable（prop 变量）, 62

W

- Web Inspector (Safari), 188–192
- web workers, 120–124
  - communication（通信）, 121
  - environment（环境）, 120
  - loading external files（加载外部文件）, 122
  - uses for（用来）, 122
- WebKit-based browsers, innerHTML, 38
- while loops（while 循环）, 62, 63
- with statements（with 语句）, 21

X

- XHR (XMLHttpRequest)
  - about（关于）, 126
  - MXHR, 128–131
  - POST versus GET（POST 对比 GET）, 127
  - sending data（发送数据）, 131–134
- XHR objects（XHR 对象）, IE, 148
- XML data format（XML 数据格式）, 134–137
- XMLHttpRequest, 131, 148
- XMLHttpRequest Script Injections（XMLHttpRequest 脚本注入）, 9
- XPath, 137

Y

- yielding with timers（使用定时器让出时间片段）, 111–120
- YSlow, 198
- YUI 3, loading JavaScript（加载 JavaScript）, 12
- YUI Profiler, 179–182

关于作者

Nicholas C. Zakas 是一位软件工程师，专注于用户界面设计以及用 JavaScript、动态 HTML、CSS、XML 和 XSLT 实现 Web 应用。他目前就职于 Yahoo! 首页小组，职位为首席前端工程师。他同时也是 YUI（Yahoo! 用户界面类库）的代码贡献者，作品有 Cookie 工具集，Profiler 和 YUI Test。

Nicholas 是《JavaScript 高级程序设计》[译注1]一书的作者，同时也是《Ajax 高级程序设计》[译注2]的合著者（均为 Wrox 出版）。此外，他还为 WebReference、Sitepoint 和 YUI Blog 撰写了大量网络文章。

Nicholas 定期发表关于 Web 开发、JavaScript 和最佳实践的演讲。他去演讲过的公司有：Yahoo!、LinkedIn、Google 和 NASA，还有 Ajax Experience、the Rich Web Experience 和 Velocity 等会议。

Nicholas 旨在通过写作和演讲来分享他在开发界上最流行的 Web 应用的工作中所学到的各种有价值信息。

想获取更多信息，可访问他的个人站点：*http://www.nczonline.net/about/*。

关于封面

本书封面上的动物是短耳猫头鹰（Asio flammeus）。正如其名，这类鸟的耳朵非常小，看似大脑袋上的山脊。它的绺十分显眼，这种猫头鹰在感觉到威胁时会进入防御状态。这是一只中等大小的猫头鹰，它有着黄色的眼睛，浅棕色且斑驳的羽毛，胸部有条纹，它宽大的翅膀上有暗色块。

它是世界上分布最广的鸟类物种之一，是一种候鸟，生存在除澳大利亚和南极洲以外的地方。短耳猫头鹰生活在原野中，比如草原、湿地或苔原。它捕获小型哺乳动物比如田鼠（和偶尔飞过的鸟），或者通过低空飞行或栖息在短树枝上等候捕获潜在猎物。这类猫头鹰活跃在黎明、傍晚和黄昏。

短耳猫头鹰飞行频率与蝙蝠或蛾相比，来回动作缓慢，翅膀拍动也不规律。在繁殖季节，雄性求偶的方式是在空中一起拍动它们强壮的翅膀，飞向高空，并迅速向地面俯冲。短耳猫头鹰同样也像是动物世界的演员——它会装死以躲避搜捕，或假装一个残缺的翅膀以吸引天敌远离巢穴。

[译注1] 《JavaScript 高级程序设计》第 2 版已由人民邮电出版社于 2010 年 7 月 1 日出版。

[译注2] 《Ajax 高级程序设计》第 2 版已由人民邮电出版社于 2008 年 7 月 1 日出版。

[General Information]
书名=高性能JavaScript
作者=（美）泽卡斯著
页数=211
SS号=13854206
DX号=
出版日期=2015.08
出版社=北京电子工业出版社

- 封面
- 书名
- 版权
- 前言
- 目录
- 第1章　加载和执行
  - 脚本位置
  - 组织脚本
  - 无阻塞的脚本
    - 延迟的脚本
    - 动态脚本元素
    - XMLHttpRequest脚本注入
    - 推荐的无阻塞模式
  - 小结
- 第2章　数据存取
  - 管理作用域
    - 作用域链和标识符解析
    - 标识符解析的性能
    - 改变作用域链
    - 动态作用域
    - 闭包、作用域和内存
  - 对象成员
    - 原型
    - 原型链
    - 嵌套成员
    - 缓存对象成员值
  - 小结
- 第3章　DOM编程
  - 浏览器中的DOM
    - 天生就慢
  - DOM访问与修改
    - innerHTML对比DOM方法
    - 节点克隆
    - HTML集合
    - 遍历DOM
  - 重绘与重排

    - 重排何时发生
    - 渲染树变化的排队与刷新
    - 最小化重绘和重排
    - 缓存布局信息
    - 让元素脱离动画流
    - IE和：hover
  - 事件委托
  - 小结
- 第4章　算法和流程控制
  - 循环
    - 循环的类型
    - 循环性能
    - 基于函数的迭代
  - 条件语句
    - if-else对比switch
    - 优化if-else
    - 查找表
  - 递归
    - 调用栈限制
    - 递归模式
    - 迭代
    - Memoization
  - 小结
- 第5章　字符串和正则表达式
  - 字符串连接
    - 加（+）和加等（+=）操作符
    - 数组项合并
    - String.prototype.concat
  - 正则表达式优化
    - 正则表达式工作原理
    - 理解回溯
    - 回溯失控
    - 基准测试的说明
    - 更多提高正则表达式效率的方法
    - 何时不使用正则表达式
  - 去除字符串首尾空白

    - 使用正则表达式去首尾空白
    - 不使用正则表达式去除字符串首尾空白
    - 混合解决方案
  - 小结
- 第6章　快速响应的用户界面
  - 浏览器UI线程
    - 浏览器限制
    - 多久才算“太久”
  - 使用定时器让出时间片段
    - 定时器基础
    - 定时器的精度
    - 使用定时器处理数组
    - 分割任务
    - 记录代码运行时间
    - 定时器与性能
  - Web Workers
    - Worker运行环境
    - 与Worker通信
    - 加载外部文件
    - 实际应用
  - 小结
- 第7章　Ajax
  - 数据传输
    - 请求数据
    - 发送数据
  - 数据格式
    - XML
    - JSON
    - HTML
    - 自定义格式
    - 数据格式总结
  - Ajax性能指南
    - 缓存数据
    - 了解Ajax类库的局限
  - 小结
- 第8章　编程实践

- 避免双重求值（Double Evaluation）
- 使用Object/Array直接量
- 避免重复工作
  - 延迟加载
  - 条件预加载
- 使用速度快的部分
  - 位操作
  - 原生方法
- 小结

第9章　构建并部署高性能JavaScript应用

- Apache Ant
- 合并多个JavaScript文件
- 预处理JavaScript文件
- JavaScript压缩
- 构建时处理对比运行时处理
- JavaScript的HTTP压缩
- 缓存JavaScript文件
- 处理缓存问题
- 使用内容分发网络（CDN）
- 部署JavaScript资源
- 敏捷JavaScript构建过程
- 小结

第10章　工具

- JavaScript性能分析
- YUI Profiler
- 匿名函数
- Firebug
  - 控制台面板分析工具
  - Console API
  - 网络面板
- IE开发人员工具
- Safari Web检查器（Web Inspector）
  - 分析面板
  - 资源面板
- Chrome开发人员工具
- 脚本阻塞

    Page Speed
    Fiddler
    YSlow
    dynaTrace Ajax Edition
    小结
索引